TURING 图灵程序设计丛书 Web开发系列

Apress®

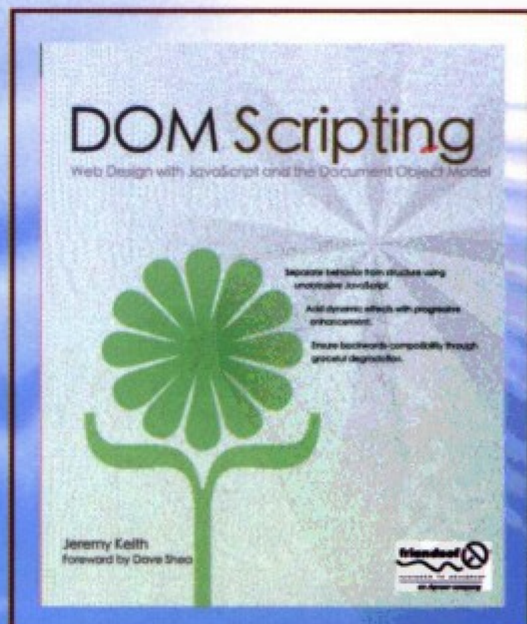
DOM Scripting

Web Design with JavaScript and the Document Object Model

JavaScript DOM 编程艺术

[英] Jeremy Keith 著

杨涛 王建桥 杨晓云 等译



- Amazon 超级畅销书
- 从 JavaScript 到 Ajax 的必由之路
- 释放 JavaScript 和 DOM 编程的惊人潜力

人民邮电出版社
POSTS & TELECOM PRESS

VISIT…

LANZAROTE Caliente.com

DOM Scripting

Web Design with JavaScript and the Document Object Model

JavaScript DOM 编程艺术

“本书理应奉为经典。文笔清晰，采用了优秀的编程原则，严格遵守相应的标准。真是完美的结合。”

——Slashdot

“我要隆重推荐本书。它前所未有地演示了 DOM 脚本编程的真正潜力。无论你是 JavaScript 新手还是专家，本书都绝对必读。”

—— Garrett Dimon，Digital-Web.com 杂志专栏作家

随着 Ajax 的兴起，JavaScript 再一次成为炙手可热的技术。然而，在历史上，它并不是一直这么幸运，由于主流浏览器之间的不兼容，以 JavaScript 为核心的 DHTML 曾经昙花一现，很快被人遗忘。

俱往矣，如今的网页设计已经翻开了新的一页。在 CSS 彻底改变了 Web 页面布局的方式之后，万维网联盟跨浏览器的 DOM 标准的制定，使 JavaScript 终于突破瓶颈，成了大大改善网页用户体验的利器。

本书在简洁明快地讲述 JavaScript 和 DOM 的基本知识之后，通过几个实例演示了大师级的网页开发技术，并透彻阐述了一些至关重要的 JavaScript 编程原则和最佳实践，包括预留退路、循序渐进和以用户为中心等。读者可以非常直观地加以领悟，迅速使自己的编程技术更上一层楼。



Jeremy Keith 国际知名的 Web 设计师，Web 标准计划（webstandards.org）成员，DOM Scripting 任务组负责人之一。除本书外，他还正在撰写众所期待的讲述 Hijax 技术的著作 *Bulletproof Ajax*（中文版将由人民邮电出版社出版）。可以通过其个人网站 adactio.com 与他联系。

Apress®

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者 / 作者热线：(010)88593802
反馈 / 投稿 / 推荐信箱：contact@turingbook.com

分类建议 计算机 / 网页制作 /JavaScript

人民邮电出版社网址 www.ptpress.com.cn

ISBN 978-7-115-13921-4

9 787115 139214 >

定价：39.00 元

TURING 图灵程序设计丛书

JavaScript DOM编程艺术

DOM Scripting

Web Design with JavaScript and the Document Object Model

[英] Jeremy Keith 著

杨涛 王建桥 杨晓云 等译

人民邮电出版社

北 京

图书在版编目（CIP）数据

JavaScript DOM 脚本编程艺术 / （英）基思著；杨涛，王建桥，杨晓云译.
—北京：人民邮电出版社，2007.1
（图灵程序设计丛书）

ISBN 978-7-115-13921-4

Ⅰ. J… Ⅱ. ①基…②杨…③王…④杨… Ⅲ. JAVA 语言—程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2006）第 133751 号

内 容 提 要

本书讲述了 JavaScript 和 DOM 的基础知识，但重点放在 DOM 编程技术背后的思路和原则：预留退路、循序渐进和以用户为中心等，这些概念对于任何前端 Web 开发工作都非常重要。本书将这些概念贯穿在书中的所有代码示例中，使你看到用来创建图片库页面的脚本、用来创建动画效果的脚本和用来丰富页面元素呈现效果的脚本，最后结合所讲述的内容创建了一个实际的网站。

本书适合 Web 设计师和开发人员阅读。

图灵程序设计丛书

JavaScript DOM 编程艺术

- 著　[英] Jeremy Keith
  译　杨 涛　王建桥　杨晓云　等
  责任编辑　傅志红
- 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
  邮编　100061　电子函件　315@ptpress.com.cn
  网址　http://www.ptpress.com.cn
  北京顺义振华印刷厂印刷
  新华书店总店北京发行所经销
- 开本：800×1000　1/16
  印张：19.75
  字数：463 千字　2007 年 1 月第 1 版
  印数：1–5 050 册　2007 年 1 月北京第 1 次印刷

著作权合同登记号　图字：01-2006-6445 号

ISBN 978-7-115-13921-4/TP・4908

定价：39.00 元

读者服务热线：(010)88593802　印装质量热线：(010)67129223

站在巨人的肩上

Standing on Shoulders of Giants

TURING

图灵教育

www.turingbook.com

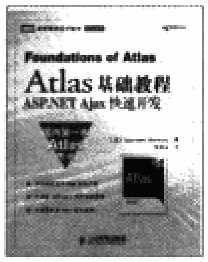
人民邮电出版社
POSTS & TELECOM PRESS

精选大师名作
打造专业品牌

TURING 图灵教育

人民邮电出版社图灵公司专注于引进国外经典教材与优秀科技图书，出版国内高校教师编写的专业教材，专业方向包括：计算机、数学、统计学和电子电气等。合作伙伴包括Prentice-Hall、Addison-Wesley、Wiley、McGraw-Hill、剑桥大学出版社、Elsevier、IEEE Press、Wrox、SIAM等多家世界知名出版公司，具有丰富的选题资源和雄厚的出版力量。

目录、前言等出版物详细信息，请浏览www.turingbook.com



书　　名：Atlas基础教程——ASP.NET Ajax快速开发
原 书 名：Foundations of Atlas: Rapid Ajax Development with ASP.NET 2.0
作　　者：Laurence Moroney
译　　者：陈黎夫
书　　号：7-115-15321-3
定　　价：39.00元
出版时间：2006年10月



书　　名：JavaScript高级程序设计
原 书 名：Professional JavaScript for Web Developers
作　　者：Nicholas C.Zakas
译　　者：曹力 张欣等
书　　号：7-115-15209-8
定　　价：59.00元
出版时间：2006年9月



书　　名：精通CSS：高级Web标准解决方案
原 书 名：CSS Mastery: Advanced Web Standards Solutions
作　　者：Andy Budd
译　　者：陈剑瓯
书　　号：7-115-15316-7
定　　价：39.00元
出版时间：2006年10月



书　　名：Oracle9i&10g编程艺术：深入数据库体系结构
原 书 名：Expert Oracle Database Architecture: 9i and 10g Programming Techniques and Solutions
作　　者：Thomas Kyte
译　　者：苏金国 王小振等
书　　号：7-115-15032-X
定　　价：99.00元
出版时间：2006年8月



书　　名：设计模式解析(第2版)
原 书 名：Design Patterns Explained: A New Perspective on Object-Oriented Design
作　　者：Alan Shalloway, James R.Trott
译　　者：徐言声
书　　号：7-115-15095-8
定　　价：45.00元
出版时间：2006年8月



书　　名：.NET设计规范
原 书 名：Framework Design Guidelines: Conventions,Idioms,and Patterns for Reusable.NET Libraries
作　　者：Krzysztof Cwalina, Brad Abrams
译　　者：葛子昂
书　　号：7-115-14929-1
定　　价：49.00 元
出版时间：2006年7月

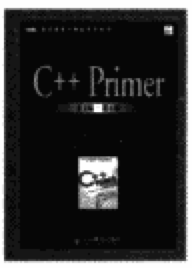


书　　名：Ajax高级程序设计
原 书 名：Professional Ajax
作　　者：NicholasC.Z akas, Jeremy McPeak, Joe Fawcett
译　　者：徐锋
书　　号：7-115-14867-8
定　　价：49.00元
出版时间：2006年7月



书　　名：Ajax实战
原 书 名：Ajax in Action
作　　者：Dave Crane, Eric Pascarello, Darren James
译　　者：ajaxcn.org
书　　号：7-115-14717-5
定　　价：69.00元
出版时间：2006年4月





大连理工大学双语教材

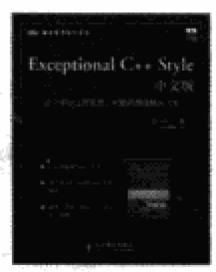
书　　名：C++Primer(第4版)(中英文版)
原 书 名：C++ Primer
作　　者：StanleyB.Lippman,Josee Lajoie, Barbara E.Moo
译　　者：李师贤 蒋爱军 梅晓勇 林瑛
中文版书号：7-115-14151-7
英文版书号：7-115-15169-7
中文版定价：99.00元
英文版定价：99.00元
中文版出版时间：2006年3月
英文版出版时间：2006年10月



书　　名：Ajax基础教程
原 书 名：Foundations of Ajax
作　　者：Ryan Asleson
译　　者：金灵 等
书　　号：7-115-14481-8
定　　价：35.00元
出版时间：2006年2月



书　　名：JSP高级程序设计
原 书 名：Beginning JavaServer Pages
作　　者：Vivek Chopra, Jon Eaves, Rupert Jones, Sing Li, John T.Bell
译　　者：朱涛江 张文静
书　　号：7-115-14522-9
定　　价：55.00元
出版时间：2006年3月



书　　名：Exceptional C++ Style中文版
原 书 名：Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions
作　　者：Herb Sutter
译　　者：刘未鹏
书　　号：7-115-14225-4
定　　价：39.00元
出版时间：2005年12月



书　　名：C++编程规范：101条规则、准则与最佳实践
原 书 名：C++ Coding Standards: 101 Rules, Guidelines, and Best Practices
作　　者：Herb Sutter, Andrei Alexandrescu
译　　者：刘基诚
书　　号：7-115-14205-X
定　　价：35.00元
出版时间：2005年12月



书　　名：Java解惑
原 书 名：Java Puzzlers: Traps, Pitfalls, and Corner Cases
作　　者：Joshua Bloch, Neal Gafter
译　　者：陈昊鹏
书　　号：7-115-14241-6
定　　价：39.00元
出版时间：2005年12月



书　　名：JSP程序设计
原 书 名：Beginning JavaServer Pages
作　　者：VivekChopra, Jon Eaves, Rupert Jones
译　　者：张文静 林琪
书　　号：7-115-14152-5
出版时间：2005年12月



书　　名：C++ 必知必会
原 书 名：C++ Common Knowledge: Essential Intermediate Programming,1E
作　　者：Stephen C.Dewhurst
译　　者：荣耀
书　　号：7-115-14101-0
定　　价：29.00元
出版时间：2005年11月

版 权 声 明

Original English language edition，entitled *DOM Scripting: Web Design with JavaScript and the Document Object Model* by Jeremy Keith , Cameron Moll , Simon Collison, published by Apress L.P., 2560 Ninth Street, Suite 219, Berkeley, CA 94710 USA.

Copyright © 2006 by Jeremy Keith. Simplified Chinese-language edition copyright © 2006 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由 Apress L.P.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

谨以此书献给我的妻子和第一位读者 Jessica。

译 者 序

网上的生活越来越丰富多彩。从最初的(X)HTML 网页，到一度热炒的 DHTML 概念，再到近几年流行起来的 CSS，网站和网页的设计工作变得越来越简便，网上的内容越来越富于变化和色彩。但是，很多网页设计者和网民朋友都不太喜欢 JavaScript，这主要有以下几方面原因。第一，很多网页设计者认为 JavaScript 的可用性很差——早期的浏览器彼此很少兼容，如果想让自己编写出来的 JavaScript 脚本在多种浏览器环境里运行，就必须编写许多用来探测浏览器的具体品牌和具体版本的测试及分支代码（术语称之为“浏览器嗅探”代码)。这样的脚本往往到处是 `if...else` 语句，既不容易阅读，又不容易复查和纠错，更难以做到让同一个脚本适用于所有的浏览器。第二，对广大的网民来说，JavaScript 网页的可访问性很差——浏览器会时不时地弹出一个报错窗口甚至导致系统死机，让人乘兴而来、败兴而去。第三，JavaScript 被很多网站用来实现弹出广告窗口的功能，人们厌烦这样的广告，也就“恨”屋及乌地厌烦起 JavaScript 来了。第四，“JavaScript”这个名字里的“Java”往往让人们误以为其源于 Java 语言，而实际接触之后才发现它们根本没有任何联系。与 Java 语言相比，JavaScript 语言要简单得多。很多程序员宁肯钻研 Java，也不愿意去了解 JavaScript 的功能和用法。

不管什么原因，JavaScript 曾经不受欢迎的确是一个事实。

现在，情况发生了极大的变化。因为几项新技术的出现，JavaScript 的春天似乎来了。首先，W3C（万维网联盟）推出的标准化 DOM（Document Object Model，文档对象模型）已经一统江湖，目前市场上常见的浏览器可以说没有不支持的。这对网页设计者来说意味着可以用简单的“对象检测”代码来取代那些繁复的浏览器嗅探代码，而按照 DOM 编写出来的 JavaScript 页面不像过去那样容易出问题，这对网民来说意味着浏览体验变得流畅了。其次，最近兴起的 Ajax 技术以 DOM 和 JavaScript 语言（以及 CSS 和 XHTML）为基本要素，基于 Ajax 技术的网站离不开 JavaScript 和 DOM 脚本。

其实，人们对 JavaScript 的恶劣印象在很大程度上来源于早期的程序员对这种语言的滥用。如果程序员在编写 JavaScript 脚本的时候能够把问题考虑得面面俱到，就可以避免许多问题，但可惜的是如此优秀的程序员太少了。事实上，即使是在 JavaScript 已经开始流行起来的今天，如果程序员在编写 JavaScript 脚本的时候不遵守相关的标准和编程准则，也仍会导致各种各样的问题。

在 2002 年前后，CSS 也是一种不太受人们欢迎的 Web 显示语言，除了用它来改变一下字体，几乎没有人用它来干其他的事情。但没过多久，人们对利用 CSS 设计网页布局的兴趣就一发而不可收，整个潮流也从那时扭转了过来。现在，掌握 CSS 已经成为许多公司在招聘网站开发人员时的一项要求。

目前，DOM 编程技术的现状与 CSS 技术在 2002 年时的境况颇有几分相似。受 Google Maps 和 Flickr 等著名公司利用 DOM 编程技术推出的 Gmail、Google Suggest 等新型服务的影响和带动，对 DOM 编程人才的需求正日益增加。有越来越多的人开始迷上了脚本编程技术并开始学习如何利用 DOM 技术去改善而不是妨碍网站的可用性和可访问性。

本书的作者 Jeremy Keith 是 Web 标准计划 DOM Scripting 任务组的负责人之一，他在这本书里通过大量示例证明了这样一个事实：只要运用得当，再注意避开那些“经典的”JavaScript 陷阱，DOM 编程技术就可以成为 Web 开发工具箱里又一件功能强大甚至是不可或缺的好东西。

本书并不是一本参考大全类型的图书，作者只重点介绍了几种最有用的 DOM 方法和属性。本书的精华在于作者在书中提到的关于 JavaScript 和 DOM 脚本编程工作的基本原则、良好习惯和正确思路。如果读者能通过书中的几个案例真正领悟这些原则、习惯和思路，就一定能让自己的编程技术再上一个台阶。

这是一本非常实用的好书，是一本值得一读再读的好书。作为本书的译者，我们相信它会让每位读者、自建网站的设计者和来到自建网站的访问者都受益匪浅。

参加本书翻译的人员还有韩兰、李京山、胡晋平、高文雅。

译 者

2006 年 9 月

序

Dave Shea

"JavaScript？别提了，用它编写程序特别麻烦。靠它建立起来的网站也不好用，动不动就弹出一个报错窗口什么的。你要是用了它，说不定它会趁你没看见的时候踢你的狗一脚呢。"

这就是 JavaScript 给我留下的印象……

作为一名 Web 设计师（或者称开发人员），我对 JavaScript 的态度是能不用就不用，你是不是也和我一样呢？像我们这样的人肯定还不少。从".com"盛极一时的年代开始，因为过时的网络教程中有太多质量低劣的示例脚本，所以人们对 JavaScript 产生抵触情绪，这根本不让我感到意外。

幸运的是，有一群像 Jeremy Keith 这样的人在努力地为人们指明正确的方向。在这本书中，他向我们证明了事情并不像我们想像得那么糟糕；只要运用得当，再注意避开那些传统的 JavaScript 陷阱，DOM 脚本编程技术就可以成为 Web 开发工具箱中又一件功能强大甚至是不可或缺的好东西。

事实上，DOM 脚本编程技术的现状让我想起了 2002 年前后的 CSS。在那之前，CSS 一直被人们认为是一种古怪的 Web 显示语言，除了用它来改变字体，几乎没有什么人用它来干其他事情。

但没过多久，人们对利用 CSS 设计网页布局的兴趣就一发而不可收了，整个潮流也从那时扭转了过来。Wired 和 ESPN 等著名企业用 CSS 重新设计网站的做法改变了人们的旧思想。我也在 2003 年初加入了向 Web 设计人员推广 CSS 技术的 CSS Zen Garden 组织。到了那年年底，CSS 已经从少数人的个人爱好变成了许多公司对网站开发人员的一种预期和要求。

现在，我们看到 DOM 脚本编程技术也正呈现出一种类似的上升趋势。受 Google Maps 和 Flickr 等著名公司在最近利用 DOM 脚本编程技术推出的新型服务的影响和带动，对 DOM 脚本编程人才的需求正在日益增加。与过去相比，有越来越多像你和我这样的人开始迷上了脚本编程技术，并开始学习如何利用 DOM 的力量增强而不是妨碍网站的可用性。

我们是幸运的，因为现在有这本书来指导我们。我也想像不出还有什么人能够比 Jeremy Keith 更适合做我们的领路人。作为 Web 标准计划 DOM Scripting 任务组的台柱，他一直站在脚本编程技术领域各种最新研发方向的最前端。再说，我从他那里"偷学"代码已经有好几年了，手中有

了这本书，我也就用不着再不好意思了。

这的确是一本值得一读再读的好书。在磕磕绊绊地看懂了前几段示例代码之后，我已经迫不及待地想看到更多的例子了。学完第 1 章后，我已经完全被吸引住了。Jeremy Keith 是极少数能把高深的概念用简明易懂的语言解释透彻的天才，他的著作不仅可以让我们知道应该怎么做，还能让我们明白为什么要那样做。

是抛开“浏览器嗅探”（browser sniffing）技术而拥抱“对象检测”（object detection）技术的时候了。再也不用假设你们网站的访问者都已经激活了 JavaScript 支持功能了。让我们舍弃那些内嵌在 HTML 文档里的事件处理函数吧，因为我们再也不需要那样做了。Web 一天一个样，而这本书里的技术会让我们每个人都获益。

前　言

这是一本讲述一种程序设计语言的书，但它也适合 Web 设计师阅读。具体地说，本书是为那些喜欢使用 CSS 和 XHTML 并愿意遵守编程规范的 Web 设计师们编写的。

本书由代码和概念两大部分构成。不要被那些代码吓倒：我知道它们乍看起来很唬人，可一旦抓住了代码背后的概念，你们就会发现用一种新语言去阅读和编写代码并没有多么困难。学习一种新的程序设计语言看起来很难，但事实却并非如此。DOM 脚本看起来似乎比 CSS 更复杂，可一旦领悟了它的语法，你们就会发现自己又多掌握了一样功能强大的 Web 开发工具。

归根结底，代码都是思想和概念的体现。我在这里要告诉大家一个秘密：其实没人能把一种程序设计语言的所有语法和关键字都记住。如果有拿不准的地方，查阅参考书就全解决了。

本书不是一本参考大全。我将只讨论编写和运行 JavaScript 脚本所必需的最基本的语法。我的真正目的是为了让大家理解 DOM 脚本编程技术背后的思路和原则。这些思路和原则或许已经是你们早就熟悉的了：预留退路、循序渐进、以用户为中心的设计。这些概念其实对任何前端 Web 开发工作都非常重要。

这些思路贯穿在本书的所有代码示例中。你们将会看到用来创建图片库页面的脚本、用来创建动画效果的脚本和用来丰富页面元素呈现效果的脚本。如果你们愿意，完全可以把这些例子剪贴到自己的代码中，但更重要的是理解这些代码背后的“如何”和“为什么”。

如果你们已经在使用 CSS 和 XHTML 来把设计思路转化为活生生的网页，就应该知道 Web 标准有多么的重要。还记得你们第一次意识到自己根本不必使用<font>标签时感受到的震撼吗？还记得你们是在何时发现自己只需修改一个 CSS 文件就可以改变整个网站的视觉效果吗？DOM 技术有着同样强大的威力。

不过，能力越大，责任也就越大。因此，我不仅想让你们看到用 DOM 脚本实现的超酷效果，更想让你们看到怎样才能利用 DOM 脚本编程技术以一种既方便自己更体贴用户的方式去充实和完善你们的网页。

如果需要本书所讨论的相关代码示例的完整清单[1]，到 http://www.friendsofed.com 网站搜索本书的主页就可以查到。你们还可以在这个网站找到 friends of ED 出版社出版的其他好书，它

1. 本书代码示例也可从图灵网站（www.turingbook.com）下载。——编辑注

们的内容涉及 Web 标准、Flash、DreamWeaver 以及许多细分的计算机领域。

你们对 JavaScript 的热情不应该在合上本书时就冷却下来。我已经在 http//domscripting.com/处开设了一个网站，我将在那里继续与大家共同探讨现代的、标准化的 JavaScript。我希望你们能到该网站看看。与此同时，我更希望本书能够对大家有所帮助。祝你们好运！

致谢

没有我的朋友和同事 Andy Budd[1]（www.andybudd.com）和 Richard Rutter（www.clagnut.com）的帮助，本书的面世就无从谈起。Andy 在我们的家乡 Brighton 开设了一个名为 Skillswap（www.skillswap.org）的免费培训网站。在 2004 年 7 月，Richard 和我在那里做了一次关于 JavaScript 和 DOM 的联合演讲。演讲结束后，我们来到附近的一家小酒馆，在那里，Andy 建议我把演讲的内容扩展成一本书。

我接受了这个想法，并就此事请教了 friends of ED 出版社的 Chris Mills。Chris 非常支持我的想法，完全没有顾虑到我以前从未写过书的事实。friends of ED 的每个人一直都在帮助和鼓励我。我要特别感谢我的项目经理 Beckie Stones 和我的文字编辑 Julie Smith 对我这个初出茅庐的作者给予的支持和谅解。

如果没有两方面的帮助，我大概永远也学不会编写 JavaScript 代码。一方面是几乎每个 Web 浏览器里都有的“view source”（查看源代码）选项。谢谢你，“view source”。另一方面是那些多年来一直在编写让人叹为观止的代码并解说重要思路的 JavaScript 大师们。Scott Andrew、Aaron Boodman、Steve Champeon、Peter-Paul Koch、Stuart Langridge 和 Simon Willison 只是我现在能想到的几位。感谢你们所有人让我分享你们的聪明才智。

感谢 Molly Holzschlag 与我分享她的经验和忠告，感谢她对本书初稿给予反馈意见。感谢 Derek Featherstone 与我多次愉快地讨论 JavaScript 问题，我喜欢他思考和分析问题的方法。

我还要特别感谢 Aaron Gustafson，他在我写作本书期间向我提供了许多宝贵的反馈和灵感。

在写作本书期间，我有幸参加两次非常棒的盛会：在得克萨斯州 Austin 举办的“South by Southwest”和在伦敦举办的@media。我要感谢这两次盛会的组织者 Hugh Forrest 和 Patrick Griffiths，是他们让我有机会结识那么多最友善的人——我从没想过我能有机会与他们结为朋友和同事。

最后，我要感谢我的妻子 Jessica Spengler，这不仅是因为她一直在默默地支持我，更因为她对本书初稿做出的专业帮助。谢谢你，我的人生伴侣。

1. Andy Budd 是 Amazon 超级畅销书《精通 CSS》的作者，该书已由人民邮电出版社出版。

目　录

第 1 章　JavaScript 简史……1
1.1　JavaScript 的起源……2
1.2　浏览器之争……3
1.2.1　DHTML……3
1.2.2　浏览器之间的冲突……4
1.3　制定标准……5
1.3.1　浏览器以外的考虑……5
1.3.2　浏览器之争的结局……6
1.3.3　新的开始……6
1.4　小结……7
第 2 章　JavaScript 语法……9
2.1　准备工作……9
2.2　语法……11
2.3　语句……11
2.4　变量……13
2.4.1　数据类型……15
2.4.2　数组……17
2.5　操作……20
2.6　条件语句……22
2.6.1　比较操作符……23
2.6.2　逻辑操作符……24
2.7　循环语句……25
2.7.1　while……25
2.7.2　do...while……26
2.7.3　for……27
2.8　函数……27
2.9　对象……31
2.9.1　内建对象……32
2.9.2　宿主对象……33
2.10　小结……33
第 3 章　DOM……35
3.1　文档：DOM 中的"D"……35
3.2　对象：DOM 中的"O"……35
3.3　模型：DOM 中的"M"……36
3.3.1　节点……38
3.3.2　getElementById()方法……42
3.3.3　getElementsByTagName()方法……44
3.4　趁热打铁……46
3.4.1　getAttribute()方法……46
3.4.2　setAttribute()方法……48
3.5　小结……49
第 4 章　案例研究：JavaScript 美术馆……51
4.1　编写标记语言文档……52
4.2　编写 JavaScript 函数……54
4.2.1　DOM 之前的解决方案……55
4.2.2　showPic()函数的代码清单……56
4.3　JavaScript 函数的调用……56
4.4　对 JavaScript 函数进行功能扩展……59
4.4.1　childNodes 属性……59
4.4.2　nodeType 属性……60
4.4.3　在 HTML 文档里增加一段描述性文本……61
4.4.4　用 JavaScript 代码改变<p>元素的文本内容……62
4.4.5　nodeValue 属性……63
4.4.6　firstChild 和 lastChild 属性……63
4.4.7　利用 nodeValue 属性刷新<p>元素的文本内容……64
4.5　小结……68
第 5 章　JavaScript 编程原则和良好习惯……69
5.1　不要误解 JavaScript……69
5.1.1　不要归罪于 JavaScript……70
5.1.2　Flash 的遭遇……71
5.1.3　质疑……72
5.2　预留退路……72

5.2.1 “javascript:”伪协议······74
5.2.2 内嵌的事件处理函数······74
5.2.3 有何不好······74
5.3 向 CSS 学习······75
5.4 分离 JavaScript······78
5.5 向后兼容性······80
5.6 小结······82

第 6 章 案例研究：JavaScript 美术馆改进版······83

6.1 快速回顾······84
6.2 解决“预留退路”问题······84
6.3 解决“分离 JavaScript”问题······85
6.3.1 添加事件处理函数······86
6.3.2 进行必要的检查······87
6.3.3 创建必要的变量······89
6.3.4 创建循环······90
6.3.5 完成必要的操作······91
6.3.6 完成 JavaScript 函数······91
6.3.7 把多个 JavaScript 函数绑定到 onload 事件处理函数上······92
6.4 JavaScript 函数的优化：不要做太多的假设······94
6.4.1 不放过每个细节······97
6.4.2 键盘浏览功能······99
6.4.3 慎用 onkeypress 事件处理函数······100
6.4.4 把 JavaScript 与 CSS 结合起来······102
6.5 DOM Core 和 HTML-DOM······105
6.6 小结······106

第 7 章 动态创建 HTML 内容······109

7.1 document.write()方法······110
7.2 innerHTML 属性······111
7.3 DOM 提供的方法······114
7.3.1 createElement()方法······115
7.3.2 appendChild()方法······116
7.3.3 createTextNode()方法······117
7.4 重回“JavaScript 美术馆”······121
7.4.1 insertBefore()方法······123
7.4.2 “JavaScript 美术馆”二次改进版······127
7.5 小结······131
7.6 下章简介······132

第 8 章 充实文档的内容······133

8.1 不应该做的事情······133
8.2 把“不可见”变成“可见”······134
8.3 原始内容······135
8.4 XHTML 文档······137
8.5 CSS······138
8.6 JavaScript 代码······139
8.6.1 显示“缩略词语表”······139
8.6.2 显示“文献来源链接表”······152
8.6.3 显示“快速访问键清单”······158
8.7 小结······161
8.8 下章简介······162

第 9 章 CSS-DOM······163

9.1 三位一体的网页······163
9.1.1 结构层······163
9.1.2 表示层······164
9.1.3 行为层······164
9.1.4 分离······165
9.2 style 属性······165
9.2.1 样式信息的检索······167
9.2.2 设置样式信息······172
9.3 何时该用 DOM 脚本去设置样式信息······174
9.3.1 根据元素在节点树里的位置来设置样式信息······174
9.3.2 根据某种条件来设置样式信息······178
9.3.3 对事件做出响应······182
9.4 className 属性······184
9.5 小结······188

第 10 章 用 JavaScript 实现动画效果······191

10.1 何为动画······191
10.1.1 位置······192
10.1.2 时间······194
10.1.3 抽象化······198
10.2 实用的动画······205

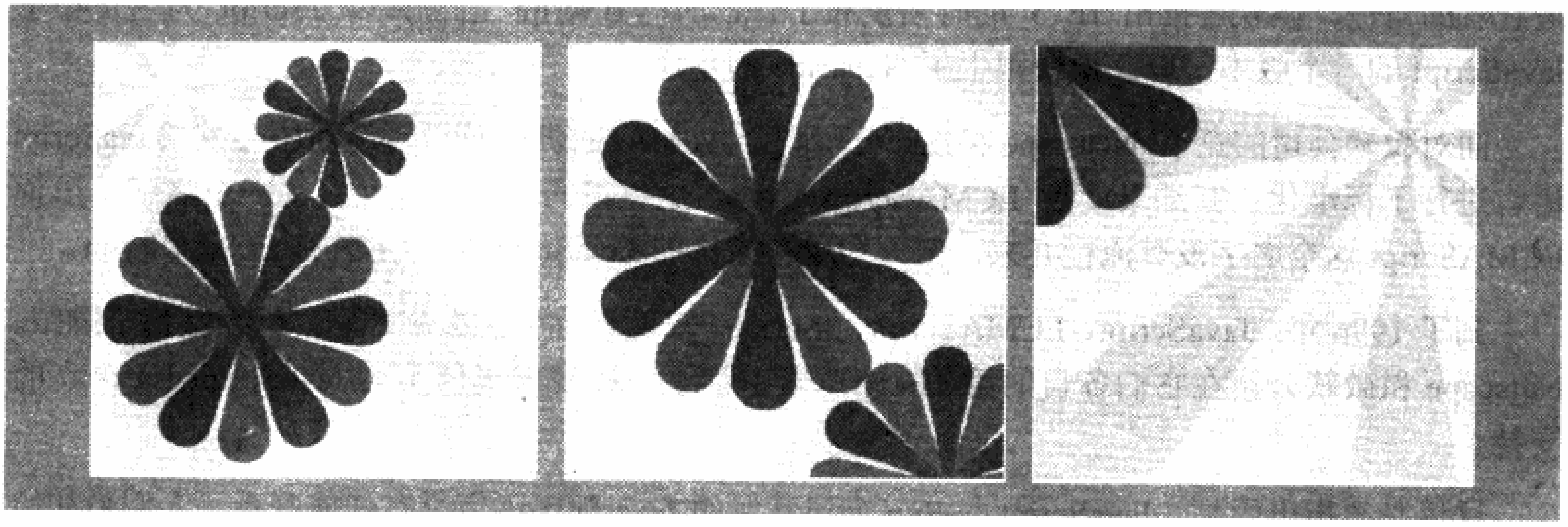
10.2.1 问题的提出 ···· 205
10.2.2 问题的解决 ···· 207
10.2.3 CSS ···· 208
10.2.4 JavaScript 代码 ···· 210
10.2.5 与变量的作用域有关的问题 ···· 214
10.3 改进动画效果 ···· 216
10.4 最后的优化 ···· 219
10.5 小结 ···· 224

第 11 章 学以致用：JavaScript 网站设计实战 ···· 225
11.1 案例背景介绍 ···· 225
11.1.1 原始材料 ···· 226
11.1.2 网站的结构 ···· 226
11.1.3 网页的结构 ···· 227
11.2 页面视觉效果设计 ···· 228
11.3 CSS ···· 229
11.4 颜色 ···· 231
11.4.1 布局 ···· 232
11.4.2 字型 ···· 234
11.5 XHTML 文档 ···· 236
11.6 JavaScript 脚本 ···· 236
11.6.1 当前页面的标识 ···· 238
11.6.2 JavaScript 动画 ···· 242
11.6.3 内部浏览 ···· 247
11.6.4 JavaScript 美术馆 ···· 251
11.6.5 改进表格 ···· 254
11.6.6 改进表单 ···· 259
11.7 小结 ···· 268
11.8 下章简介 ···· 268

第 12 章 展望 DOM 脚本编程技术 ···· 269
12.1 Web 的现状 ···· 269
12.1.1 Web 浏览器 ···· 270
12.1.2 Web 设计师 ···· 271
12.1.3 三条腿的凳子 ···· 272
12.2 Ajax 技术 ···· 274
12.2.1 XMLHttpRequest 对象 ···· 275
12.2.2 Ajax 技术的爆发 ···· 278
12.2.3 循序渐进：如何运用 Ajax 技术 ···· 279
12.2.4 Ajax 技术的未来 ···· 281
12.3 Web 应用 ···· 282
12.4 小结 ···· 283

附录 DOM 方法和属性 ···· 285

第1章 JavaScript 简史



本章内容

- ❑ JavaScript 的起源
- ❑ 浏览器之争
- ❑ DOM 的演变史

对网页设计人员来说，这是一个充满着挑战和机遇的时代。近几年来，网页设计工作已经从一种混乱无序和即兴发挥的状态，逐渐发展为一种有着成熟的设计原则可供遵循的流水线作业。有越来越多的网页设计人员开始采用一种标准化的思路来建立网站，而实现这一思路和方法的具体技术则称为“Web 标准”。

当网页设计人员谈论起与 Web 标准有关的话题时，XHTML（可扩展的超文本标记语言）和 CSS（层叠样式表）通常占据着核心地位。不过，由 W3C（万维网联盟）批准并由所有与标准相兼容的 Web 浏览器支持的第三方技术称为 DOM（文档对象模型）。我们可以利用 DOM 去改善文档的可交互性，就像我们可以利用 CSS 给文档添加各种样式一样。

在开始学习 DOM 之前，我们先回顾一下使网页具备可交互性的程序设计语言。这种语言就是 JavaScript，它已经诞生相当长的时间了。

1.1 JavaScript 的起源

JavaScript 是 Netscape 公司与 Sun 公司合作开发的。在 JavaScript 出现之前，Web 浏览器不过是一种能够显示超文本文档的软件的基本部分。而在 JavaScript 出现之后，网页的内容不再局限于枯燥的文本，它们的可交互性得到了显著的改善。JavaScript 的第一个版本，即 JavaScript 1.0 版本，出现在 1995 年推出的 Netscape Navigator 2 浏览器中。

在 JavaScript 1.0 发布时，Netscape Navigator 主宰着浏览器市场，微软的 IE 浏览器则扮演着追赶者的角色。微软在推出 IE 3 的时候发布了自己的 VBScript 语言并以 JScript 为名发布了 JavaScript 的一个版本，以此很快跟上了 Netscape 的步伐。

面对微软公司的竞争，Netscape 和 Sun 公司联合 ECMA（欧洲计算机制造商协会）对 JavaScript 语言进行了标准化。其结果就是 ECMAScript 语言，这使得同一种语言又多了一个名字。虽说 ECMAScript 这个名字没有流行开来，但人们现在谈论的 JavaScript 实际上就是 ECMAScript。

到了 1996 年，JavaScript、ECMAScript、JScript——随便你们怎么称呼它，已经站稳了脚跟。Netscape 和微软公司在它们各自的第 3 版浏览器中都不同程度地提供了对 JavaScript 1.1 语言的支持。

这里必须指出的是，JavaScript 与 Sun 公司开发的 Java 程序语言没有任何联系。人们最初给 JavaScript 起的名字是 LiveScript，后来选择“JavaScript”作为其正式名称的原因，大概是想让它听起来有系出名门的感觉，但令人遗憾的是，这一选择反而更容易让人们把这两种语言混为一谈，而这种混淆又因为各种 Web 浏览器确实具备这样或那样的 Java 客户端支持功能的事实被进一步放大和加剧。事实上，虽说 Java 在理论上几乎可以部署在任何环境中，但 JavaScript 却只局限于 Web 浏览器。

JavaScript 是一种脚本语言，JavaScript 脚本通常只能通过 Web 浏览器去完成某种操作而不是像普通意义上的程序那样可以独立运行。因为需要由 Web 浏览器进行解释和执行，所以 JavaScript 脚本不像 Java 和 C++等编译型程序设计语言那样用途广泛。不过，这种相对的简单性也正是 JavaScript 的长处：因为比较容易学习和掌握，所以 JavaScript 很受那些本身不是程序员，但希望能够通过简单的剪贴操作把脚本嵌入他们的现有网页中的普通用户们的欢迎。

JavaScript 还向程序员提供了一种操控 Web 浏览器的手段。例如，JavaScript 语言可以用来调整 Web 浏览器窗口的高度、宽度和屏显位置等属性。以这种办法给出 Web 浏览器本身的属性可以看做是 BOM（浏览器对象模型）。JavaScript 的早期版本还提供了一种初级的 DOM（文档对象模型）。

什么是 DOM

简单地说，DOM 是一套对文档的内容进行抽象和概念化的方法。

在现实世界里，人们对笔者称之为“世界对象模型”里的许多事物都有一个共同的理解。例如，当用“汽车”、“房子”和“树”等名词来称呼日常生活环境里的事物时，我们几乎可以百分之百地肯定对方知道我们说的是什么，而这是因为人们对这些名词所代表的具体事物都有着同样的认识。于是，当对别人说“汽车停在了车库里”时，可以相当有把握地假设他们不会把这句话理解为“小鸟关在了壁橱里”。

我们的“世界对象模型”不仅可以用来描述客观存在的事物，还可以用来描述各种抽象概念。

例如，假设有个人向我问路，而我给出的答案是“左边第三栋房子”。这个答案有没有意义将取决于那个人能否理解“第三”和“左边”等抽象概念的含义。如果他不会数数或者分不清左右，则不管他是否理解这几个概念，我的回答对他都不会有任何帮助。在现实世界里，正是因为大家对抽象的世界模型有着基本的共识，人们才能用非常简单的话把比较复杂的含义表达出来并得到对方的理解。具体到这里的例子，我可以相当有把握地断定，那位老兄以及其他人对“第三”和“左边”等抽象概念的理解和我对这些概念的理解是完全一样的。

这个道理对网页也同样适用。JavaScript 的早期版本向程序员提供了对 Web 文档的某些实际内容（主要是图像和表单）进行查询和操控的手段。因为“图像”和“表单”等名词是程序员都明白的概念，JavaScript 语言也预先定义了“images”和“forms”等关键字，我们才能像下面这样在 JavaScript 脚本里引用“文档中的第三个图像”或“文档中名为‘details’的表单”：

```
document.images[2]
document.forms['details']
```

现在的人们通常把这种试验性质的初级 DOM 称为“第 0 级 DOM”（DOM Level 0）。在还未形成统一标准的初期阶段，“第 0 级 DOM”的常见用法包括对图像进行链接和显示以及在客户端进行某种形式的数据合法性验证。但从 Netscape 和微软公司各自推出的第四代浏览器产品开始，DOM 受到了越来越多的开发人员和爱好者的关注。

1.2 浏览器之争

Netscape Navigator 4（NN 4）浏览器发布于 1997 年 6 月，IE 4 浏览器发布于同年的 10 月。这两种浏览器都对它们的早期版本进行了许多改进，使用得到极大扩展的 DOM，可以通过 JavaScript 完成的功能大大增加，而网页设计人员也开始熟悉一个新的名词：DHTML。

1.2.1 DHTML

DHTML 是“dynamic HTML”（动态 HTML）的简称。严格地说，DHTML 并不是一项单一的新技术，而是 HTML、CSS 和 JavaScript 这三种技术相结合的产物。DHTML 背后的含义是：

- 利用 HTML 把网页标记为各种元素；
- 利用 CSS 设计各有关元素的排版样式并确定它们在窗口中的显示位置；
- 利用 JavaScript 实时地操控和改变各有关样式。

DHTML 指的是上述三项技术的相互结合。利用 DHTML，复杂的动画效果一下子变得非常容易实现。例如，可以先用 HTML 标记一个如下所示的页面元素：

```
<div id="myelement">This is my element</div>
```

然后，可以用 CSS 为这个页面元素定义如下所示的位置样式：

```
#myelement {
  position: absolute;
  left: 50px;
  top: 100px;
}
```

接下来，只需利用 JavaScript 改变 myelement 元素的 left 和 top 样式，就可以让它在页面上随意移动。

不过，这种简单性只停留在理论上——因为 NN 4 和 IE 4 浏览器使用的是不同的且不兼容的 DOM，所以要想实际获得这种效果还需要程序员做很多工作。换句话说，虽然浏览器制造商的目标是一样的，但他们在解决 DOM 问题时采用的办法却完全不同。

1.2.2　浏览器之间的冲突

Netscape 公司的 DOM 使用了其专有的元素，这些元素称为*层*（layer）。这些层都有唯一的 ID，JavaScript 代码需要像下面这样使用它们：

```
document.layers['myelement']
```

而在微软公司的 DOM 中这个元素必须像下面这样使用：

```
document.all['myelement']
```

这两种 DOM 在细节方面的差异并不止于这一点。假设你想找出 myelement 元素的 left 位置并把它赋值给变量 xpos，那么在 Netscape Navigator 4 浏览器里必须这样做：

```
var xpos = document.layers['myelement'].left;
```

而在 IE 4 浏览器中，你需要使用如下所示的语句才能完成同样的工作：

```
var xpos = document.all['myelement'].leftpos;
```

这就导致了一种很可笑的局面：程序员在编写 DOM 脚本代码时必须知道它们将运行在哪种浏览器环境里。在实际工作中，许多脚本都不得不编写两次，一次为 NN 4 浏览器，另一次为 IE 4 浏览器。同时，为了确保能够正确地向不同的浏览器提供与之相应的脚本，程序员还必须编写一些代码去检测在客户端运行的浏览器到底是哪一种。

DHTML 打开了一扇通往新世界的大门，但进入这扇大门的人们却发现这条路并不好走。因此，没多久，DHTML 就从一个大热门变成了一个人们不愿提起的名词，而对这种技术的评价也很快地变成了“宣传噱头”和“难以实现”。

1.3 制定标准

就在浏览器的制造商们为了压倒竞争对手而以 DOM 为武器展开一场营销大战的同时，W3C 不事声张地推出了一个标准化的 DOM。令人欣慰的是，Netscape、微软和其他一些浏览器制造商们还能抛开彼此的敌意而与 W3C 携手制定新的标准，并于 1998 年 10 月完成了“第 1 级 DOM”（DOM Level 1）。

回到刚才的例子，我们一起看看新的标准化 DOM 是如何解决同样的问题的。我们已经用 `<div>`标签定义了一个 ID 为 `myelement` 的页面元素，现在需要找出它的 `left` 位置并把这个值保存到变量 `xpos` 中。下面是需要用到的语法：

```
var xpos = document.getElementById('myelement').style.left
```

乍看起来，这与刚才那两种非标准化的专有 DOM 相比并没有明显的改进。但事实上，标准化的 DOM 有着非常远大的抱负。

浏览器制造商们感兴趣的只不过是一些通过 JavaScript 操控网页的具体办法，但 W3C 推出的标准化 DOM 却可以让任何一种程序设计语言对使用任何一种标记语言编写出来的任何一份文档进行操控。

1.3.1 浏览器以外的考虑

DOM 是一种 API（应用编程接口）。简单地说，API 就是一组已经得到有关各方共同认可的基本约定。在现实世界中，相当于 API 的例子包括（但不限于）：

- 摩尔斯码
- 国际时区
- 化学元素周期表

以上这些都是不同学科领域中的标准，它们使得人们能够更方便地进行交流与合作。如果没有这样的标准，事情往往会演变成为一场灾难。别忘了，英制度量衡与公制度量衡之间的差异至少导致过一次火星探测任务的失败。

在软件编程领域中，虽然存在着多种不同的语言，但很多任务却是相同或相似的。这也正是人们需要 API 的原因。一旦掌握了某个标准，就可以把它应用在许多不同的环境中。虽然有关的语法会因为使用的程序设计语言而有所变化，但这些约定却总是保持不变的。

因此，在学完这本关于如何通过 JavaScript 使用 DOM 的书之后，你们获得的关于 DOM 的新知识对你们今后的工作——例如，需要使用诸如 PHP 或 Python 之类的程序设计语言去解析一份 XML 文档的时候，也会有很大的帮助。

W3C 对 DOM 的定义是：“一个与系统平台和编程语言无关的接口，程序和脚本可以通过这个接口动态地对文档的内容、结构和样式进行访问和修改。”

W3C 推出的标准化 DOM，在独立性和适用范围等诸多方面，都远远超出了各自为战的浏览器制造商们推出的各种专有 DOM。

1.3.2 浏览器之争的结局

我们知道，浏览器市场份额大战的赢家是微软公司，但具有讽刺意味的是，专有的 DOM 和 HTML 标记对这个最终结果并无影响。IE 浏览器之所以能击败其他对手，其主要原因不过是所有运行着 Windows 操作系统的个人电脑都预装了它而已。

受浏览器之争影响最重的人群是那些网站和网页设计人员。需要同时支持多种浏览器的软件开发工作，曾经是程序员们的噩梦。除了刚才提到的那些在 JavaScript 实现方面的差异之外，Netscape Navigator 和 IE 这两种浏览器在对 CSS 的支持方面也有许多非常不同的地方。有不少程序员都把编写那些可以同时工作在这两种浏览器环境下的样式表和脚本的工作视为一种黑色艺术。

为了打破浏览器制造商们筑起的专利壁垒，一群志同道合的程序员建立了名为 Web 标准计划（简称 WaSP，`http://webstandards.org/`）的小组。WaSP 小组采取的第一个行动就是，鼓励浏览器制造商们接受 W3C 制定和推荐的各项标准——也就是在浏览器制造商们的帮助下得以起草和完善的那些标准。

或许是因为来自 WaSP 小组的压力，又或许是因为企业的内部决策，浏览器制造商后来推出的下一代浏览器产品对 Web 标准的支持得到了极大的改善。

1.3.3 新的开始

微软公司在 IE 5 浏览器中内建了对 W3C 制定的标准化 DOM 的支持，但同时继续支持其专有的 Microsoft DOM。

Netscape 公司采取的是一种更为坚决果断的做法，它们发布了一种与 NN 4 无任何共同点的浏览器：Netscape Navigator 6（NN 6），该浏览器干脆跳过了主版本号并使用一个与以前完全不同的呈现引擎，新引擎对 CSS 提供了更多更好的支持。NN 6 也支持标准化的 DOM，但不再向后兼容早期的 Netscape DOM。

之后，Netscape 和微软公司都发布过几个新版浏览器产品，而每个新版本都在老版本的基础上通过增加对各项 Web 标准的支持来不断完善自己。但令人遗憾的是，微软公司的浏览器开发工作在它们推出 IE 6 之后停顿了下来，这使得 IE 6 中的 CSS 实现还存在一些问题没能得到彻底解决。尽管如此，它对“第 1 级 DOM”的支持还是非常坚定的。

与此同时，其他一些浏览器也开始崭露头角。Apple 公司在 2003 年发布了 Web 浏览器 Safari，它对 DOM 标准的支持完全没有问题。Firefox、Mozilla 和 Camino 等浏览器对 DOM 的支持也非常完善，它们都采用了与 Nescape 6 和 Nescape 7 完全一样的开源呈现引擎。Opera 和

Konquerer 浏览器也提供了对 DOM 的良好支持。

目前使用的 95%以上的浏览器都具备对 DOM 的内建支持。发生在 20 世纪 90 年代后期的浏览器大战已经离我们越来越遥远。虽说还没有一种浏览器能够提供对 W3C DOM 的完美支持，但现代的浏览器都至少实现了 W3C 相关标准中 95%的规范，而这意味着在编写 JavaScript 代码时几乎不需要考虑它们将运行在何种浏览器环境下了。

虽然 IE 浏览器的开发工作停顿了下来，但网页设计人员的日子已经不像过去那么困难了。过去，程序员在编写 JavaScript 脚本时往往不得不增加一些代码去探测在客户端运行的是哪种浏览器；现在，程序员只需编写一次代码就几乎可以把它们应用在所有的场合了。只要遵守 DOM 标准，程序员就可以相当有把握地确信，自己编写的脚本几乎在所有的环境下都可以正常工作。

1.4 小结

在前面对 JavaScript 发展简史的介绍中，笔者特别提到，不同的浏览器采用了不同的办法来完成同样的任务。这一无法回避的事实不仅主宰着如何编写 JavaScript 脚本代码，还影响着 JavaScript 教科书的编写方式。

JavaScript 教科书的作者往往会提供大量的示例代码以演示这种脚本语言的使用方法，而完成同一项任务的示例脚本往往需要为不同的浏览器编写两次或更多次。就像你们在绝大多数网站上查到的代码一样，在绝大多数 JavaScript 教科书的示例脚本中往往充斥着大量的浏览器检测代码和分支调用结构。类似地，在 JavaScript 技术参考文献中，函数和方法的清单也往往是一式多份——至少需要标明哪种浏览器支持哪些函数和方法。

如今这种情况已经有所改变。多亏了标准化的 DOM，不同的浏览器在完成同样的任务时采用的细节做法已经非常一致了。因此，在本书中，当演示如何使用 JavaScript 和 DOM 完成某项任务时，将不再需要撇开主题去探讨如何对付不同的浏览器。

如果无特殊的必要，本书将尽量避免涉及任何一种特定的浏览器。

此外，我们在本书后面的内容中将不再使用“DHTML”这个术语，因为我们认为这个术语与其说是一个技术性词汇，不如说是一个市场营销噱头。首先，它听起来很像是 HTML 或 XHTML 语言的另一种扩展，因而很容易造成误解或混淆；其次，这个术语容易勾起一些痛苦的回忆——如果你向 20 世纪 90 年代后期的程序员们提起“DHTML”，你将很难让他们相信它现在已经变成了一种简单、易用的标准化技术。

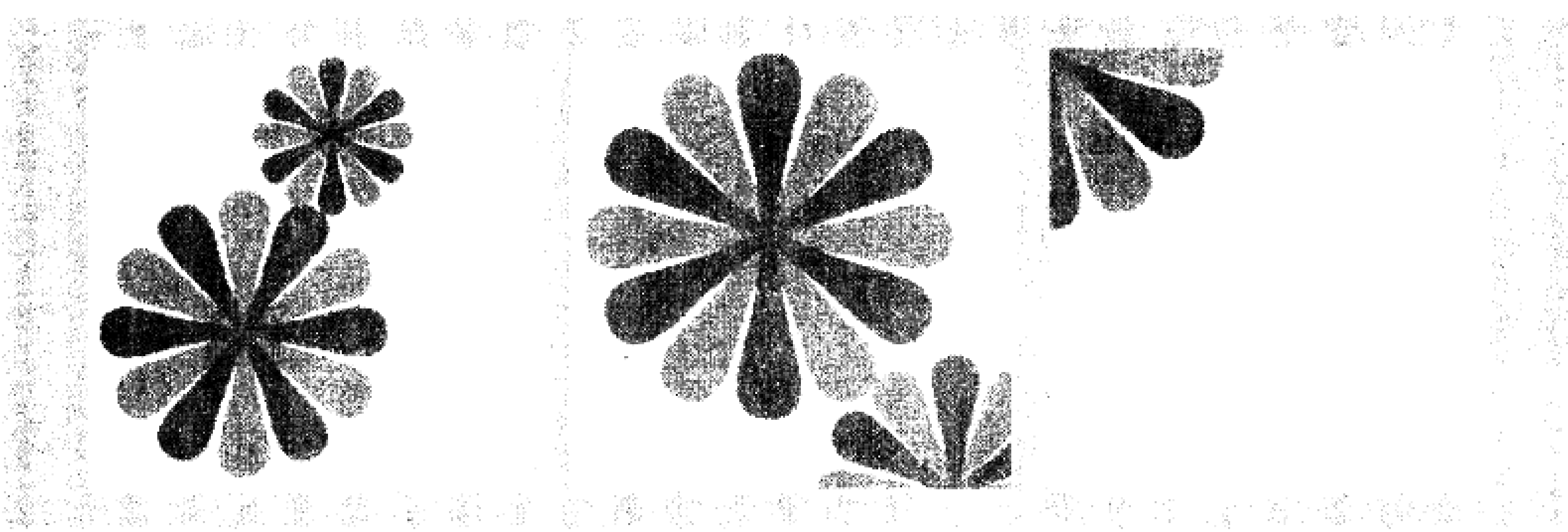
DHTML 是 HTML/XHTML、CSS 和 JavaScript 相结合的产物，但把这些东西真正凝聚在一起的是 DOM。如果真的需要有个词汇来描述这一过程的话，我们就应该使用一个更精确的词汇。用 DHTML 来称呼与浏览器有关的编程工作并不是不可以，但用它来描述基于有关标准的代码编写工作就不那么恰当了。在探讨如何使用 W3C DOM 来处理文档和样式表时，我们认为“DOM 脚本程序设计”是一种更精确的说法。

DHTML 只适用于 Web 文档，“DOM 脚本程序设计”则涵盖了使用任何一种支持 DOM API 的程序设计语言去处理任何一种标记文档的所有情况。具体到 Web 文档，JavaScript 语言的特点使它成为了 DOM 脚本程序设计的最佳选择。

在正式介绍 DOM 脚本程序设计技巧之前，我们将在下一章先向大家简要地介绍一下 JavaScript 的语法。

第 2 章

JavaScript 语法



本章内容

- ❑ 语句
- ❑ 变量和数组
- ❑ 操作符
- ❑ 条件语句和循环语句
- ❑ 函数与对象

本章将对 JavaScript 语法中最重要的一些概念进行简要的介绍。

2.1 准备工作

编写 JavaScript 脚本不需要任何特殊的软件，一个普通的文本编辑器和一个 Web 浏览器就足够了。

用 JavaScript 编写的代码必须嵌在一份 HTML/XHTML 文档中才能得以执行。这可以通过两种办法做到。第一种办法是将 JavaScript 代码插入文档<head>部分的<script>标签间，如下所示：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
<head>
<script type="text/javascript">
JavaScript goes here...
```

```
</script>
</head>
<body>
Mark-up goes here...
</body>
</html>
```

更好的办法是先把 JavaScript 代码存入一个独立的文件——建议把.js 作为这种文件的扩展名，再利用<script>标签的 src 属性指向该文件，如下所示：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html>
<head>
<script type="text/javascript" src="file.js">
</script>
</head>
<body>
Mark-up goes here...
</body>
</html>
```

如果有兴趣试用一下本章中的示例，可以用文本编辑器先创建两个文件。首先，创建一个简单的 HTML 或 XHTML 文档框架，这个文件可以命名为诸如 test.html 之类的名称。这里的要点是，在这份文档的<head>部分包含一个<script>标签，该标签的 src 属性设置为用文本编辑器将要创建的第二个文件的名字，比如 example.js。

test.html 文件应该包含如下所示的内容：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
 <head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Just a test</title>
  <script type="text/javascript" src="example.js">
  </script>
</head>
<body>
</body>
</html>
```

可以把本章中的任何一个示例拷贝到 example.js 文件中。虽说那些示例没有什么特别令人激动的地方，但它们可以把有关的语法演示得明明白白。

在本书后面的章节里，将演示如何使用 JavaScript 改变文档的行为和内容。但就这一章的示例脚本而言，我们将只使用一个简单的对话框来显示消息。

如果改变了 example.js 文件的内容，只需在 Web 浏览器中重新加载 test.html 文档即可查

看到其执行效果。Web 浏览器将立刻以解释方式执行 JavaScript 代码。

程序设计语言分为解释型和编译型两大类。Java 或 C++等语言需要一个**编译器**（compiler）。编译器是一种能够把用 Java 等高级语言编写出来的源代码翻译为直接在计算机上执行的二进制可执行文件的程序。

解释型程序设计语言不需要编译器——它们仅需要解释器。对于 JavaScript 语言，在 WWW 环境下，Web 浏览器将负责完成有关的解释和执行工作。浏览器中的 JavaScript 解释器将直接读入源代码并加以执行。如果没有解释器，JavaScript 代码将无法得到执行。

如果用编译型程序设计语言编写的代码包含着错误，那些错误在代码编译阶段就会被发现并报告。对于解释型程序设计语言，代码中的错误只有等到解释器实际执行到有关代码时才会被发现并报告。

与解释型程序设计语言相比，编译型程序设计语言往往速度更快，可移植性也更好，但它们的学习曲线往往相当陡峭。

JavaScript 的优点之一是比较容易学习和掌握，但千万不要因此而小看 JavaScript 能力：它能够完成许多相当复杂的编程任务。不过，本章将只介绍它的最基本的语法和用途。

2.2 语法

英语是一种解释型的语言。在阅读和处理别人用英语写出来的文字时，阅读者本人就相当于一个英语解释器。只要作者遵守了英语的语法规则，他想表达的意思就可以被阅读者正确地解读出来。**语法**（syntax）的广义含义包括语句、单词、标点符号等各个方面，它的狭义含义则特指语句结构方面的各项规则。在接下来的讨论中，我们使用的是“语法”这个词的狭义含义。

与那些有文字的人类语言一样，每种程序设计语言都有自己的语法。JavaScript 语言的语法与 Java 和 C++等其他一些程序设计语言的语法非常相似。

2.3 语句

用 JavaScript 或任何一种其他程序设计语言编写出来的脚本都是由一系列指令构成的，这些指令称为**语句**（statement）。只有按照正确的语法编写出来的语句才能得到正确的解释。

JavaScript 语句与英语中的句子很相似。它们是任何一个脚本的基本构成单位。

英语语法要求每个句子必须以一个大写字母开头、以一个句号结尾。JavaScript 在这方面的要求不那么严格，程序员只需简单地把各条语句放在不同的行上就可以分隔它们，如下所示：

```
first statement
second statement
```

如果你想把多条语句放在同一行上，就必须像下面这样用分号来分隔它们：

```
first statement; second statement;
```

即使没有把多条语句放在同一行上，但在每条语句的末尾加上一个分号，也是一种良好的编程习惯：

```
first statement;
second statement;
```

这可以让代码更容易阅读。将每条语句单独占用一行的做法可以让你本人或其他程序员更容易追踪 JavaScript 脚本的执行流程。

注释

JavaScript 解释器并不要求 JavaScript 脚本中的每条语句都必须是可执行的。有时，需要在脚本中写出一些仅供参考或提示性的信息，但并不希望 JavaScript 解释器真的去执行这样的语句。这种语句称为注释（comment）。

注释语句非常有用，它们可以让你把编写代码时的一些想法和考虑记载下来供今后参考，还可以帮助你追踪有关代码的执行流程。类似于日常生活中的便条，注释语句可以帮助程序员跟踪和追查在脚本中发生的事情。

有多种在 JavaScript 脚本中插入注释的具体做法。例如，如果使用了两个斜杠作为一行的开头，这一行就将被解释为一条注释：

```
// Note to self: comments are good.
```

如果使用这种记号方式，就必须在每行注释的开头加上两个斜杠。也就是说，像下面这样的做法是有问题的——第 2 行将不会被解释为一条注释：

```
// Note to self:
   comments are good.
```

如果你想写出两行注释，就必须把它们写成如下所示的样子：

```
// Note to self:
// comments are good.
```

一条跨越多行的注释还可以用下面这个方式来给出：在整段注释内容的开头加上一个“/*”，在整段注释内容的末尾加上一个“*/”。下面是一个多行注释的例子：

```
/* Note to self:
   comments are good  */
```

这种记号方式在需要插入跨越多行的大段注释内容时很有用，它可以提高整个脚本的可读性。

还可以使用 HTML 风格的注释，但这种做法仅适用于单行注释。换句话说，JavaScript 解释器对“<!–”的处理与对“//”的处理是一样的：

```
<!- This is a comment in JavaScript.
```

如果是在 HTML 文档中，还需要以“ –>”来结束这种注释语句，如下所示：

```
<!- This is a comment in HTML ->
```

但 JavaScript 不要求这样做，它会把“ ->”视为注释内容的一部分。

请注意，HTML 允许上面这样的注释跨越多个行，但 JavaScript 要求这种注释的每行都必须在开头加上“<!-”来作为标志。

因为 JavaScript 解释器在处理这种风格的注释时与大家所熟悉的 HTML 做法不同，为避免发生混淆，笔者建议大家最好不要在 JavaScript 脚本中使用 HTML 风格的注释。如果没有特别的理由，用“//”记号给出单行注释、用“/*”记号给出多行注释。

2.4 变量

在日常生活里，有些东西是固定不变的，有些东西则会发生变化。例如，人的姓名和生日是固定不变的，但心情和年龄却会随着时间的推移而发生变化。在谈论程序设计语言时，人们把那些会发生变化的东西称为变量（variable）。

我的心情会随着我的切身感受而变化。假设我有一个变量 mood（意思是“心情”），我可以把此时此刻的心情存放到这个变量中。不管这个变量的值是“happy”还是“sad”，它的名字始终是 mood。我可以随时改变这个值。

类似地，假设我现在的年龄是 33 岁。一年之后，我的年龄就是 34 岁。我可以使用变量 age 来存放我的年龄并在生日那天改变这个值。当我现在去查看 age 变量时，它的值是 33；但一年之后，它的值将变成 34。

把值存入变量的操作称为赋值（assignment）。我把变量 mood 赋值为“happy”，把变量 age 赋值为 33。

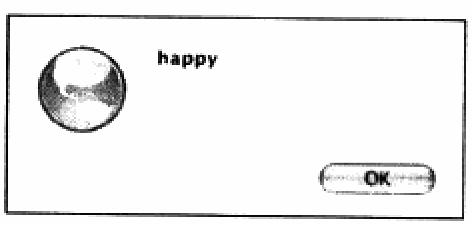
下面是在 JavaScript 中对这些变量进行赋值的语法：

```
mood = "happy";
age = 33;
```

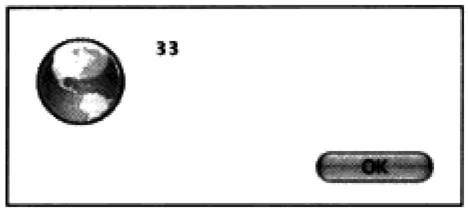
把值赋值给变量后，我们就可以说该变量包含这个值。例如，变量 mood 现在包含值“happy”，变量 age 现在包含值 33。我们可以用如下所示的语句把这两个变量的值显示在一个弹出式警告窗口中：

```
alert(mood);
alert(age);
```

下面是一个显示 mood 变量值的例子。



下面是一个显示 age 变量值的例子。



我们将在本书后面的章节中用变量做一些有用的事情，请大家耐心地阅读下去。

请注意，JavaScript 允许程序员直接对变量进行赋值而无需提前对它们做出声明。这在许多程序设计语言中都是不允许的。有相当一部分程序设计语言要求在使用变量之前必须先对它做出“介绍”——术语称之为*声明*（declare）。

在 JavaScript 脚本中，如果程序员在对某个变量进行赋值之前未对其做出声明，赋值操作将自动声明该变量。虽然 JavaScript 没有要求程序员必须这么做，但提前对变量做出声明仍是一种良好的编程习惯。下面的语句对变量 mood 和 age 做出了声明：

```
var mood;
var age;
```

每次只声明一个变量的做法并不是绝对的，JavaScript 也允许程序员用一条语句声明多个变量，如下所示：

```
var mood, age;
```

JavaScript 甚至允许程序员把声明变量和对该变量进行赋值的这两项操作合起来一次完成：

```
var mood = "happy";
var age = 33;
```

我们甚至还可以像下面这样做：

```
var mood = "happy", age = 33;
```

像上面这样声明和赋值各有关变量是最有效率的做法，这条语句的效果相当于下面这些语句的总和：

```
var mood, age;
mood = "happy";
age = 33;
```

在 JavaScript 语言里，变量和其他语法元素的名字都是区分字母大小写的。名字是 mood 的变量与名字是 Mood、MOOD 或 mOOd 的变量没有任何关系，它们不是同一个变量。下面的语句是在对两个不同的变量进行赋值：

```
var mood = "happy";
MOOD = "sad";
```

JavaScript 语法不允许变量的名字中包含空格或标点符号（但美元符号“$”例外）。下面这条语句将导致语法错误：

```
var my mood = "happy";
```

JavaScript 变量名允许包含字母、数字、美元符号和下划线字符。为了让比较长的变量名有更好的可读性，可以在变量名中的适当位置插入一个下划线字符，就像下面这样：

```
var my_mood = "happy";
```

在上面这条语句中，单词“happy”是 JavaScript 语言中的一个*字面量*（literal），也就是可以在 JavaScript 代码中直接写出来的数据内容。字面量除了它本身所给出的内容外无任何附加含义，用大力水手 Popeye 的话来说：“它就是它！”与此形成对照的是，单词“var”是 JavaScript 语言中的一个关键字，my_mood 是一个变量的名字；它们都有自身内容以外的含义。

2.4.1 数据类型

变量 mood 的值是一个*字符串类型的字面量*，变量 age 的值则是一个*数值类型的字面量*。虽然它们是两种不同的数据类型，但在 JavaScript 脚本中为它们做出声明和进行赋值的语法无任何区别。有些程序设计语言要求程序员在声明变量的同时还必须明确地对其数据类型做出声明，这种做法称为*类型声明*（typing）。

要求程序员必须明确地对数据类型做出声明的程序设计语言被称为*强类型*（strongly typed）语言。像 JavaScript 这样不要求程序员进行类型声明的语言则被称为*弱类型*（weakly typed）语言。所谓弱类型意味着程序员可以随意改变某个变量的数据类型。

下面这条语句在强类型语言中是非法的，但在 JavaScript 语言里却完全没有问题：

```
var age = "thirty three";
age = 33;
```

JavaScript 并不关心变量 age 的值是*字符串*还是*数值*。

接下来，我们一起看看 JavaScript 语言中最重要的几种数据类型。

1. 字符串

字符串由零个或多个字符构成。字符包括字母、数字、标点符号和空格。字符串必须放在引号里——单引号或双引号都允许使用。下面这两条语句有着同样的效果：

```
var mood = 'happy';
var mood = "happy";
```

你们可以随意选用，但最好能根据字符串所包含的字符来加以选择：如果字符串包含双引号字符，就应该把整个字符串放在单引号中；如果字符串包含单引号字符，就应该把整个字符串放在双引号中：

```
var mood = "don't ask";
```

如果想在上面这条语句中使用单引号，就必须保证字母“n”和“t”之间的单引号能被解释为这个字符串的一部分。换句话说，必须保证这个单引号被解释为这个字符串里的一个字符，而不是被解释为这个字符串的结束标志。这个问题需要使用字符转义（escaping）功能来解决。在 JavaScript 语言中，对字符进行转义需要用到反斜杠字符，如下所示：

```
var mood = 'don\'t ask';
```

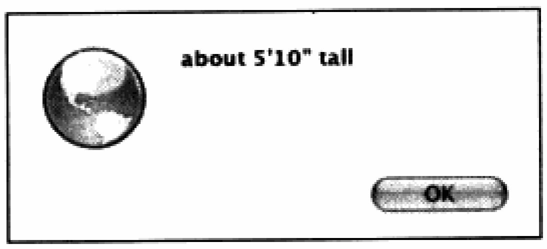
类似地，如果想用双引号来给出一个本身就包含着双引号字符的字符串，就必须用反斜杠字符对这个字符串中的双引号进行转义，如下所示：

```
var height = "about 5'10\" tall";
```

为了对其他字符进行转义而添加到字符串中的那些反斜杠字符并不是字符串的实际组成部分。你们可以自己去验证这一点：把下面这段代码添加到 example.js 文件中，然后重新加载 test.html 文件：

```
var height = "about 5'10\" tall";
alert(height);
```

下面是用反斜杠字符对有关字符进行转义后的一个屏幕输出示例。



就个人而言，笔者比较喜欢用双引号来给出字符串。作为一种良好的编程习惯，不管你们选择的是双引号还是单引号，最好能在整个脚本中保持一致。如果在同一个脚本中一会儿使用双引号，一会儿又使用单引号，代码很快就会变得难以阅读和理解。

2. 数值

如果想让某个变量包含一个数值，不用限定它必须是一个整数。JavaScript 允许程序员使用带小数点的数值，并且允许数值是任意位数，这类数值称为浮点数（floating-point number）：

```
var age = 33.25;
```

还可以使用负数。负数的表示方式是在有关数值的前面加上一个减号（–），如下所示：

```
var temperature = -20;
```

JavaScript 并不要求负数必须是一个整数：

```
var temperature = -20.33333333
```

以上这些都是数值数据类型的例子。

3. 布尔值

另一种重要的 JavaScript 数据类型是布尔（boolean）类型。

布尔数据只有两种可取值——true 或 false。假设需要这样一个变量：如果我正在睡觉，这个变量将存储一个值；如果我没有睡觉，这个变量将存储另一个值。可以用字符串数据类型来解决这个问题——只要根据具体情况把有关变量赋值为“sleeping”或“not sleeping”即可达到目的，但布尔数据类型显然是一个更好的选择：

```
var sleeping = true;
```

从某种意义上讲，为计算机设计程序就是与布尔值打交道。作为最基本的事实，所有的电子电路只能识别和使用布尔数据：电路中有电流或是没有电流。根据具体情况，这两种状态可以代表“真或假”、“是或否”或者“1 或 0”，但不管它们代表什么，这两种状态绝不可能同时出现——换句话说，在任意时刻只能使用两种可取值中的一种。

与字符串值不同，千万不要把布尔值用引号括起来。布尔值 false 与字符串值'false'或"false"是两回事！

下面这条语句将把变量 married 设置为布尔值 true：

```
var married = true;
```

下面这条语句将把变量 married 设置为一个包含着单词"true"的字符串：

```
var married = "true";
```

2.4.2 数组

字符串、数值和布尔值都属于离散值（scalar）。如果某个变量是离散的，它在任意时刻就只能有一个值。如果想用一个变量来存储一组值，就需要使用数组（array）。

数组是由名字相同的多个值构成的一个集合，集合中的每个值都是这个数组的元素（element）。例如，我们可以用名为 beatles 的变量来保存 Beatles 乐队全体四位成员的姓名。

在 JavaScriptd 脚本中，数组要用关键字 Array 来声明。在声明数组的同时，程序员还可以对这个数组的元素个数，也就是这个数组的长度（length），做出规定：

```
var beatles = Array(4);
```

有时，我们无法提前预知某个数组最终会容纳多少个元素。这没有关系，JavaScript 并不要求在声明数组时必须给出它的元素个数，我们完全可以在声明数组时不给出明确具体的元素个数：

```
var beatles = Array();
```

向数组中添加元素的操作称为填充（populating）。在填充数组时，不仅需要给出新元素的值，还需要在数组中为新元素指定存放位置，这个位置要通过下标（index）给出。数组里的每个元

素都有一个相应的下标。在语句中，下标值必须放在方括号内，如下所示：

```
array[index] = element;
```

现在来填充刚才声明的 beatles 数组。我们将按照人们在提到 Beatles 乐队成员时的传统顺序（即 John、Paul、George 和 Ringo）进行。首先是第一个下标和元素：

```
beatles[0] = "John";
```

以 0 而不是 1 作为第一个下标值多少会让人感到有些不习惯，但 JavaScript 语言就是这么规定的，所以我们这里只能这么做。这一点很重要，但也很容易被忘记，初出茅庐的程序员在刚接触 JavaScript 数组时经常在这个问题上犯错误。

下面是声明和填充 beatles 数组的全过程：

```
var beatles = Array(4);
beatles[0] = "John";
beatles[1] = "Paul";
beatles[2] = "George";
beatles[3] = "Ringo";
```

有了上面这些代码，我们即可在脚本中通过下标值“2”（beatles[2]）来检索取值为“George”的元素了。请注意，beatles 数组的长度是 4，但它最后一个元素的下标却是 3。因为数组下标是从 0 开始计数的，你们或许需要一些时间才能习惯这一事实。

像上面这样填充数组未免有些麻烦。这里有一种相对简单的方式：在声明数组的同时对它进行填充。这么做时别忘了用逗号把各个元素分隔开：

```
var beatles = Array("John","Paul","George","Ringo");
```

上面这条语句会为 beatles 数组中的每个元素自动分配一个下标：第一个下标是 0，第二个是 1，依此类推。因此，beatles[2]仍将对应于取值为“George”的元素。

我们甚至用不着明确地表明我们是在创建数组。事实上，只需用一对方括号把各个元素的初始值括起来就足以创建出我们想要的数组了：

```
var beatles = ["John","Paul","George","Ringo"];
```

不过，在声明或填充数组时写出 Array 关键字是一个良好的编程习惯，这可以提高 JavaScript 脚本的可读性，并让我们一眼就看出哪些变量是数组。

数组元素不必非得是字符串。可以把一些布尔值存入一个数组，还可以把一组数值存入一个数组：

```
var years = Array(1940,1941,1942,1943);
```

甚至可以把这三种数据类型混在一起存入一个数组：

```
var lennon = Array("John",1940,false);
```

数组元素还可以是变量：

```
var name = "John";
beatles[0] = name;
```

这将把 beatles 数组的第一个元素赋值为“John”。

数组元素的值还可以是另一个数组的元素。下面两条语句将把 beatles 数组的第二个元素赋值为“Paul”：

```
var names = Array("Ringo","John","George","Paul");
beatles[1] = names[3];
```

事实上，数组还可以包含其他的数组！数组中的任何一个元素都可以把一个数组作为它的值：

```
var lennon = Array("John",1940,false);
var beatles = Array();
beatles[0] = lennon;
```

现在，beatles 数组的第一个元素的值是另外一个数组。要想获得那个数组里的某个元素的值，我们需要使用更多的方括号。beatles[0][0]的值是“John”，beatles[0][1]的值是 1940，beatles[0][2]的值是 false。

这是一种功能相当强大的存储和获取信息的方式，但如果我们不得不记住每个下标数字的话，编程工作将是一种非常痛苦和麻烦的体验。还好，有一种办法可以让我们以更可读的方式去填充数组。

关联数组

beatles 数组是数值数组的一个典型例子：每个元素的下标是一个数字，每增加一个元素，这个数字就依次增加 1。第一个元素的下标是 0，第二个元素的下标是 1，依此类推。

如果在填充数组时只给出了元素的值，这个数组就将是一个数值数组，它的各个元素的下标将被自动创建和刷新。

我们可以通过在填充数组时为每个新元素明确地给出下标的方式来改变这种默认的行为。在为新元素给出下标时，不必局限于整数数字。数组下标可以是字符串：

```
var lennon = Array();
lennon["name"] = "John";
lennon["year"] = 1940;
lennon["living"] = false;
```

这称为关联数组（associative array）。从某种意义上讲，完全可以把所有的数组都看作是关联数组。尽管数值数组的下标是由系统自动创建的一些数字，但每个下标仍关联着一个特定的值。因此，数值数组完全可以被当作关联数组的一种特例来对待。

用关联数组来代替数值数组的做法意味着，我们可以通过各元素的名字而不是一个下标数字来引用它们。这可以大大提高脚本的可读性。

下面，我们将创建一个新的 beatles 数组，并用刚才创建的 lennon 数组来填充它的第一个元素。别忘了，数组的元素可以是另一个数组：

```
var beatles = Array();
beatles[0] = lennon;
```

现在，可以通过一些有意义的名字去访问所需要的某个元素了。beatles[0]["name"]的值是“John”，beatles[0]["year"]的值是 1940，beatles[0]["living"]的值是 false。

在此基础上，还可以做进一步的改进：把 beatles 数组也填充为关联数组而不是数值数组。这样一来，我们就可以用“drummer”或“bassist”等更有意义且更容易记忆的字符串值，而不是一些枯燥乏味的整数作为下标去访问这个数组里的元素了：

```
var beatles = Array();
beatles["vocalist"] = lennon;
```

现在，beatles["vocalist"]["name"]的值是“John”，beatles["vocalist"]["year"]的值是 1940，beatles["vocalist"]["living"]的值是 false。

2.5 操作

我们此前给出的示例语句都非常简单，只是创建了一些不同类型的变量而已。要想通过 JavaScript 去完成一些有用的工作，我们还需要能够进行计算和处理数据。这就需要完成一些操作（operation）。

算术操作符

加法是一种操作，减法、除法和乘法也是如此。这些算术操作（arithmetic operation）中的每一种都必须借助于相应的操作符（operator）才能完成。操作符是 JavaScript 为完成各种操作而定义的一些符号。你们其实已经见过一种操作符了：它就是刚才在进行赋值操作时使用的等号（=）。加法操作符是加号（+），减法操作符是减号（–），除法操作符是斜杠（/），乘法操作符是星号（*）。

下面是一个简单的加法操作：

```
1 + 4
```

还可以把多种操作组合在一起：

```
1 + 4 * 5
```

为避免产生歧义，可以用括号把不同的操作分隔开来：

```
1 + (4 * 5)
(1 + 4) * 5
```

变量可以包含操作：

```
var total = (1 + 4) * 5;
```

不仅如此，我们还可以对变量进行操作：

```
var temp_fahrenheit = 95;
var temp_celsius = (temp_fahrenheit - 32) / 1.8;
```

JavaScript 还提供了一些非常有用的操作符作为各种常用操作的快捷方式。例如，如果你想给一个数值变量加上 1，可以使用如下所示的语句：

```
year = year + 1;
```

也可以使用++操作符来达到同样的目的：

```
year++;
```

类似地，--操作符将对一个数值变量的值进行减 1 操作。

加号（+）是一个比较特殊的操作符：它既可以用于数值，也可以用于字符串。把两个字符串合二为一是一种很直观易懂的操作：

```
var message = "I am feeling " + "happy";
```

像这样把多个字符串首尾相连在一起的操作叫作拼接（concatenation）。这种拼接也可以通过变量来完成：

```
var mood = "happy";
var message = "I am feeling " + mood;
```

我们甚至可以把数值和字符串拼接在一起；因为 JavaScript 是一种弱类型语言，所以这种操作是允许的。此时，数值将被自动转换为字符串：

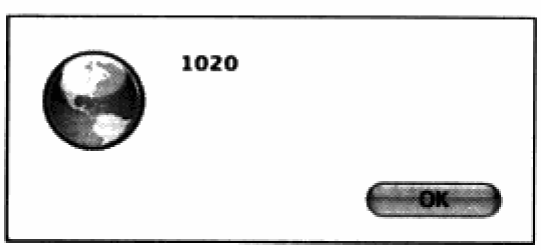
```
var year = 2005;
var message = "The year is " + year;
```

请记住，如果把字符串和数值拼接在一起，其结果将是一个更长的字符串；但如果你用同样的操作符来“拼接”两个数值，其结果将是那两个数值的算术和。请对比下面两条 alter 语句的执行结果：

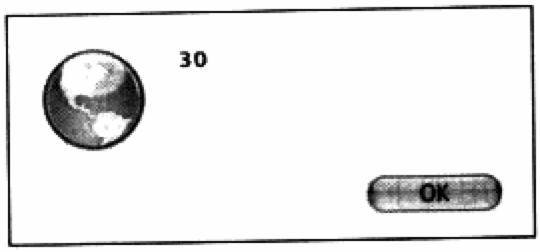
```
alert ("10" + 20);
alert (10 + 20);
```

第一条 alter 语句将返回字符串"1020"，第二条 alter 语句将返回数值 30。

下面是对字符串"10"和数值 20 进行拼接的结果。



下面是对数值 10 和数值 20 进行加法运算的结果：



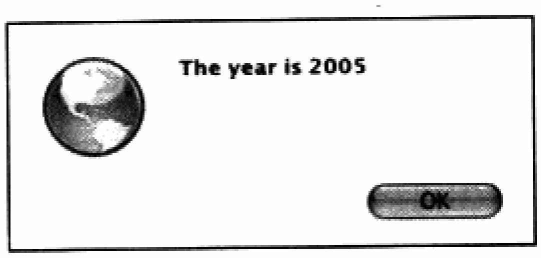
另一个非常有用的快捷操作符是+=，它可以一次完成“加法和赋值”（或“拼接和赋值”）操作：

```
var year = 2005;
var message = "The year is ";
message += year;
```

执行完上面这些语句后，变量 message 的值将是“The year is 2005”。可以用如下所示的 alter 对话框来验证这一结果：

```
alert(message);
```

这次对字符串和数值进行拼接操作的结果如下所示。



2.6 条件语句

此前介绍给大家的语句都是相对比较简单的声明或运算，而脚本的真正威力体现在它们还可以根据人们给出的各种条件做出判断和决策。JavaScript 脚本需要使用条件语句（conditional statement）来做出判断和决策。

在解释脚本时，浏览器将依次执行这个脚本中的各条语句，而我们可以在这个脚本中用条件语句来设置一个条件，只有满足了这一条件才能让更多的语句得到执行。最常见的条件语句是 if 语句，下面是 if 语句的基本语法：

```
if (condition) {
  statements;
}
```

条件必须放在 if 后面的圆括号中。条件的求值结果永远是一个布尔值，即只能是 true 或 false。花括号中的语句——不管它们有多少条，只有在给定条件的求值结果是 true 的情况下才会得到执行。因此，在下面这个例子中，alter 消息永远也不会出现：

```
if (1 > 2) {
  alert("The world has gone mad!");
}
```

因为 1 不可能大于 2，所以上面这个条件的值永远是 false。

在这条 if 语句中，我是故意把所有的东西都放在花括号里的。这并不是 JavaScript 的一项语法要求——这么做只是为了让代码更容易阅读和理解。

事实上，if 语句中的花括号本身并不是必不可少的。如果 if 语句中的花括号部分只包含着一条语句的话，那就根本用不着使用花括号，而且这条 if 语句的全部内容可以写在同一行上：

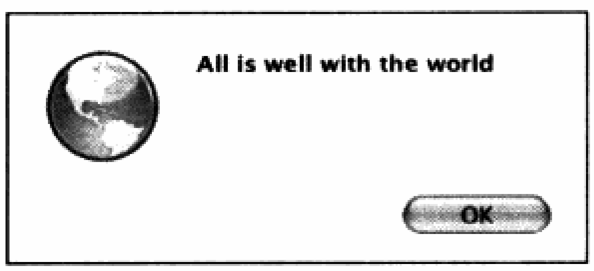
```
if (1 > 2) alert("The world has gone mad!");
```

不过，因为花括号可以提高脚本的可读性，所以在 if 语句中使用花括号通常是个好主意。

if 语句还可以用 else 来扩展。包含在 else 子句中的语句会在给定条件的值为 false 时得到执行：

```
if (1 > 2) {
  alert("The world has gone mad!");
} else {
  alert("All is well with the world");
}
```

因为给定条件“1>2”的值是 false，所以我们将看到如下所示的结果。



2.6.1 比较操作符

JavaScript 还提供了许多几乎只能用在条件语句里的操作符，其中包括诸如大于（>）、小于（<）、大于或等于（>=）、小于或等于（<=）之类的比较操作符。

如果想比较两个值是否相等，可以使用“等于”比较操作符。这个操作符由两个等号构成（==）。别忘了，单个等号（=）是用于完成赋值操作的。如果你在条件语句的某个条件里使用了单个等号，那么只要相应的赋值操作取得成功，那个条件的求值结果就将是 true。

下面是一个错误地进行“等于”比较的例子：

```
var my_mood = "happy";
var your_mood = "sad";
if (my_mood = your_mood) {
  alert("We both feel the same.");
}
```

上面这条语句的错误之处在于，它是把变量 your_mood 赋值给变量 my_mood，而不是在比较它们是否相等。因为这种赋值操作总会成功，所以这个条件语句的结果将永远是 true。

下面才是进行“等于”比较的正确做法：

```
var my_mood = "happy";
var your_mood = "sad";
if (my_mood == your_mood) {
 alert("We both feel the same.");
}
```

这次，条件语句的结果是 false。

JavaScript 还提供了一个用来进行“不等于”比较的操作符，它由一个感叹号和一个等号构成（!=）。

```
if (my_mood != your_mood) {
  alert("We're feeling different moods.");
}
```

2.6.2 逻辑操作符

JavaScript 允许我们把条件语句里的操作组合在一起。例如，如果想检查某个变量——不妨假设这个变量的名字是 num，它的值是不是在 5～10 之间，我将需要进行两次比较操作：首先，比较这个变量是否大于或等于 5；然后，比较这个变量是否小于或等于 10。这两次比较操作叫作*逻辑操作数*（operand）。下面是把这两个逻辑操作数组合在一起的具体做法：

```
if ( num>=5 && num<=10 ) {
  alert("The number is in the right range.");
}
```

在这里使用了“逻辑与”操作符，它由两个“&”字符构成（&&），它是一个逻辑操作符。

逻辑操作符的操作对象是布尔值。每个逻辑操作数返回一个布尔值 true 或者是 false。“逻辑与”操作只有在它的两个操作数都是 true 时才会是 true。

“逻辑或”操作符由两个垂直线字符构成（||）。只要它的操作数中有一个是 true，“逻辑或”操作就将是 true。如果它的两个操作数都是 true，“逻辑或”操作也将是 true。只有当它的两个操作数都是 false 时，“逻辑或”操作才会是 false。

```
if ( num > 10 || num < 5 ) {
  alert("The number is not in the right range.");
}
```

JavaScript 还提供了一个“逻辑非”操作符，它由一个感叹号（!）单独构成。“逻辑非”操作符只能作用于单个逻辑操作数，其结果是把那个逻辑操作数所返回的布尔值取反：如果那个逻辑操作数所返回的布尔值是 true，“逻辑非’操作符将把它取反为 false：

```
if ( !(1 > 2) ) {
  alert("All is well with the world");
}
```

请注意，为避免产生歧义，在上面这条语句中把逻辑操作数放在了括号里——我想让“逻辑非”操作符作用于括号里的所有内容。

我们可以用“逻辑非”操作符把整个条件语句的结果颠倒过来。在下面的例子里，我特意使用了一对括号来确保 “逻辑非”操作符将作用于两个逻辑操作数的组合结果：

```
if ( !(num > 10 || num < 5) ) {
  alert("The number IS in the right range.");
}
```

2.7 循环语句

if 语句或许是最重要、最有用的条件语句了。if 语句的唯一不足是它无法用来完成重复性的操作。在 if 语句里，包含在花括号里的代码块只能执行一次。如果需要反复多次地执行同一个代码块，就必须使用循环语句。

循环语句可以让我们反复多次地执行同一段代码。循环语句分为几种不同的类型，但它们的工作原理几乎一样：只要给定条件仍能得到满足，包含在循环语句里的代码就将重复地执行下去；一旦给定条件的求值结果不再是 true，循环也就到此为止。

2.7.1 while

while 循环与 if 语句非常相似，它们的语法几乎完全一样：

```
while (condition) {
  statements;
}
```

while 循环与 if 语句唯一的区别是：只要给定条件的求值结果是 true，包含在花括号里的代码就将反复地执行下去。下面是一个 while 循环的例子：

```
var count = 1;
while (count < 11) {
  alert (count);
  count++;
}
```

我们来仔细分析一下上面这段代码。首先，创建数值变量 count 并赋值为 1；然后，以 count < 11——意思是“只要变量 count 的值小于 11，就重复执行这个循环”，为条件创建一个 while 循环。在 while 循环的内部，用“++”操作符对变量 count 的值执行加 1 操作；而这一操作将重复执行 10 次。如果用 Web 浏览器来观察这段代码的执行情况，将会看到一个 alert 对话框闪现了 10 次。这条循环语句执行完毕后，变量 count 的值将是 11。

注意 这里的关键是在while循环的内部必须发生一些会影响循环控制条件的事情。在上例中，我们在while循环的内部对变量count的值进行了加1操作，而这将导致循环控制条件在经过10次循环后的求值结果会变成false。如果我们不对变量count的值执行加1操作，这个while循环将永远执行下去。

2.7.2 do...while

类似于 if 语句的情况，while 循环的花括号部分所包含的语句有可能不被执行：因为对循环控制条件的求值发生在每次循环开始之前，所以如果循环控制条件的首次求值结果是 false，那些代码将一次也不会被执行。

在某些场合，我们需要那些被包含在循环语句内部的代码至少执行一次。这时，do 循环将是我们的最佳选择。下面是 do 循环的语法：

```
do {
  statements;
} while (condition);
```

这与刚才介绍的 while 循环非常相似，但有个显而易见的区别：对循环控制条件的求值发生在每次循环结束之后。因此，即使循环控制条件的首次求值结果是 false，包含在花括号里的语句也至少会被执行一次。

我们可以把前一小节里的 while 循环改写为如下所示的 do...while 循环：

```
var count = 1;
do {
  alert (count);
  count++;
} while (count < 11);
```

这段代码的执行结果与 while 循环的执行结果完全一样：alter 消息将闪现 10 次；在循环结束后，变量 count 的值将是 11。

再来看看下面这个变体：

```
var count = 1;
do {
  alert (count);
  count++;
} while (count < 1);
```

在上面这个 do 循环里，循环控制条件的求值结果永远不为 true：变量 count 的初始值是 1，所以它在这里永远不会小于 1。可是，因为 do 循环的循环控制条件出现在花括号部分之后，所以包含在这个 do 循环内部的代码还是执行了一次。也就是说，仍将看到一条 alert 消息。这里还有一个细节需要大家注意：这些语句执行完毕后，变量 count 的值将是 2 而不是 1——虽然这没有改变这个例子里“循环控制条件的求值结果是 false”的事实。

2.7.3 for

用 for 循环来重复执行一些代码也很方便。从循环执行一些代码的意义上讲，它类似于 while 循环；从另一个方面看，for 循环只是刚才介绍的 do 循环的一种变体形式。如果仔细观察上一小节里的 do 循环的例子，我们就会发现它们都可以被改写为如下所示的样子：

```
initialize;
while (condition) {
  statements;
  increment;
}
```

而 for 循环不过是把如上所示的循环结构进一步改写为如下所示的紧凑形式而已：

```
for (initial condition; test condition; alter condition) {
  statements;
}
```

用 for 循环来重复执行一些代码的好处是循环控制结构更加清晰。与循环有关的所有内容都包含在 for 语句的圆括号部分。

我们可以把上一小节里的 do 循环例子改写为如下所示的 for 循环：

```
for (var count = 1; count < 11; count++ ) {
  alert (count);
}
```

与循环有关的所有内容都包含在 for 语句的圆括号里。现在，当我们把一些代码放在花括号中间的时候，我们清楚地知道那些代码将被执行 10 次。

for 循环最常见的用途之一是对某个数组里的全体元素进行遍历处理。这往往需要用到数组的 array.length 属性，这个属性可以告诉我们在给定数组里的元素的个数：

```
var beatles = Array("John","Paul","George","Ringo");
for (var count = 0 ; count < beatles.length; count++ ) {
  alert(beatles[count]);
}
```

运行这段代码，你们将看到 4 条 alert 消息，它们分别对应着 Beatles 乐队的四位成员。

2.8 函数

如果需要多次使用同一组语句，还可以把它们打包为一个函数。所谓函数（function）就是一组允许人们在代码里随时调用的语句。从效果上看，每个函数都相当于一个短小的脚本。

作为一种良好的编程习惯，你们应该先对函数做出定义再调用它们。

下面是一个简单的示例函数：

```
function shout() {
  var beatles = Array("John","Paul","George","Ringo");
  for (var count = 0 ; count < beatles.length; count++ ) {
    alert(beatles[count]);
  }
}
```

这个函数里的循环语句将依次弹出几个 alert 对话框来显示 Beatles 乐队成员的名字。现在，如果你想在自己的脚本里执行这一动作，可以随时使用如下所示的记号来调用这个函数：

```
shout();
```

每当需要反复执行一段代码时，我们都可以利用函数来避免重复输入大量的相同内容。不过，函数的真正威力体现在，我们可以把不同的数据传递给它们，而它们将使用实际传递给它们的数据去完成预定的操作。在把数据传递给函数时，我们把那些数据称为**参数**（**argument**）。

下面是用来定义一个函数的语法：

```
function name(arguments) {
  statements;
}
```

JavaScript 提供了许多可以拿来就用的内建函数，在前面的内容里多次出现过的 alert()就是一个内建函数：这个函数需要我们提供一个参数，它将弹出一个对话框来显示这个参数的值。

在定义函数时，你可以为它声明任意多个参数，只要记得用逗号把它们分隔开来就行。在函数的内部，你可以像使用普通变量那样使用它的任何一个参数。

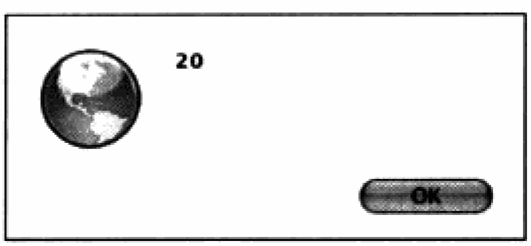
下面是一个需要传递两个参数的函数。如果我们把两个数值传递给这个函数，这个函数将对它们进行乘法运算：

```
function multiply(num1,num2) {
  var total = num1 * num2;
  alert(total);
}
```

在定义了这个函数的脚本里，我们可以从任意位置去调用这个函数，如下所示：

```
multiply(10,2);
```

把数值 10 和 2 传递给 multiply()函数的结果如下所示。



这将产生这样一种视觉效果：屏幕上会立刻弹出一个显示乘法运算结果（20）的 alert 对话框。如果这个函数能把乘法运算结果返回给调用这个函数的语句往往会更有用。这很容易做到：

函数不仅能够（以参数的形式）接收数据，还能够返回数据。

我们完全可以创建一个函数并让它返回一个数值、一个字符串、一个数组或一个布尔值。这需要用到 return 语句：

```
function multiply(num1,num2) {
  var total = num1 * num2;
  return total;
}
```

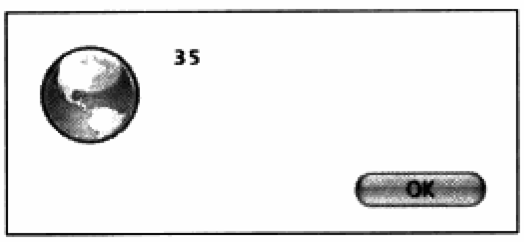
下面这个函数只有一个参数（一个华氏温度值），它将返回一个数值（同一温度的摄氏温度值）：

```
function convertToCelsius(temp) {
  var result = temp - 32;
  result = result / 1.8;
  return result;
}
```

函数的真正价值体现在，我们还可以把它们当作一种数据类型来使用，而这意味着我们可以把一个函数的调用结果赋值给一个变量：

```
var temp_fahrenheit = 95;
var temp_celsius = convertToCelsius(temp_fahrenheit);
alert(temp_celsius);
```

把华氏温度值 95 转换为摄氏温度值的结果如下所示。



在这个例子里，变量 temp_celsius 的值将是 35，这个数值由 convertToCelsius 函数返回。

你们或许想知道我是如何命名变量和函数的。在命名变量时，我用下划线来分隔各个单词；在命名函数时，我从第二个单词开始把每个单词的第一个字母写成大写形式（也就是所谓的 Camel 记号）。我这么做是为了让自己能够一眼看出哪些名字是变量、哪些名字是函数。与变量的情况一样，JavaScript 语言也不允许函数的名字里包含空格。Camel 记号可以在不违反这一规定的前提下，把变量和函数的名字以一种既简单又明确的方式区分开来。

变量的作用域

前面讲过，作为一种良好的编程习惯，在第一次对某个变量进行赋值时应该用 var 对其做出声明。当在函数内部使用变量时，就更应该这么做。

变量既可以是全局的，也可以是局部的。在谈论全局变量和局部变量之间的区别时，我们其实是在讨论变量的作用域（scope）。

全局变量（global variable）可以在脚本中的任何位置被引用。一旦你在某个脚本里声明了一个全局变量，你就可以从这个脚本中的任何位置——包括各有关函数的内部——引用它。全局变量的作用域是整个脚本。

局部变量（local variable）只存在于对它做出声明的那个函数的内部，在那个函数的外部是无法引用它的。局部变量的作用域仅限于某个特定的函数。

因此，我们在函数里既可以使用全局变量，也可以使用这个函数的局部变量。这一细节很有用，但它有时也会导致一些问题。如果在一个函数的内部不小心使用了某个全局变量的名字，即使本意是想使用一个局部变量，JavaScript 也会认为是在引用那个全局变量。

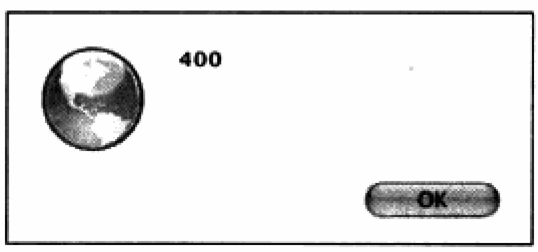
还好，我们可以用 var 关键字明确地为在函数中使用的变量设定一个作用域。

如果在某个函数中使用了 var，那个变量就将被视为一个局部变量，它将只存在于这个函数的上下文中；反之，如果没有使用 var，那个变量就将被视为一个全局变量，如果你的脚本里已经存在一个与之同名的变量，这个函数将覆盖那个现有变量的值。

我们来看下面这个例子：

```
function square(num) {
  total = num * num;
  return total;
}
var total = 50;
var number = square(20);
alert(total);
```

这些代码将不可避免地导致全局变量 total 的值发生变化。



全局变量 total 的值变成了 400。我的本意是让 square()函数只把它计算出来的平方值返回给变量 number，但因为未在这个函数里用 var 关键字把它内部的 total 变量明确地声明为局部变量，这个函数把名字同样是 total 的那个全局变量的值也改变了。

把这个函数写成如下所示的样子才是正确的：

```
function square(num) {
  var total = num * num;
  return total;
}
```

现在，全局变量 total 变得安全了，再怎么调用 square()函数也不会影响到它。

请记住，函数在行为方面应该像一个自我包容的脚本，而这意味着在定义一个函数时，我们必须把它内部的变量全都明确地声明为局部变量。如果从没忘记在函数里使用 var 关键字，就可以避免任何形式的二义性隐患。

2.9 对象

对象（object）是一种非常重要的数据类型，但此前我们还没有提到过它。对象是自我包含的数据集合，包含在对象里的数据可以通过两种形式——即属性（property）和方法（method）访问：

- 属性是隶属于某个特定对象的变量；
- 方法是只有某个特定对象才能调用的函数。

对象就是由一些彼此相关的属性和方法集合在一起而构成的一个数据实体。

在 JavaScript 脚本里，属性和方法都需要使用如下所示的“点”语法来访问：

```
Object.property
Object.method()
```

你们已经见过如何用 mood 和 age 等变量来存放诸如“心情”和“年龄”之类的值。如果它们是某个对象的属性——这里不妨假设那个对象的名字是 Person，我们就必须使用如下所示的记号来使用它们：

```
Person.mood
Person.age
```

假如 Person 对象还关联着一些诸如 walk()和 sleep()之类的函数，这些函数就是这个对象的方法，而我们必须使用如下所示的记号来访问它们：

```
Person.walk()
Person.sleep()
```

把这些属性和方法全部集合在一起，我们就得到了一个 Person 对象。换句话说，我们可以把 Person 看作是所有这些属性和方法的统称。

为了使用 Person 对象来描述一个特定的人，我们需要创建一个 Person 对象的实例（instance）。实例是对象的具体表现；对象是统称，实例是个体。例如，你和我都是人，都可以用 Person 对象来描述；但你和我是两个不同的个体，很可能有着不同的属性（例如，你和我的年龄可能不一样）。因此，你和我对应着两个不同的 Person 对象——它们虽然都是 Person 对象，但它们是两个不同的实例。

在 JavaScript 语言里，为给定对象创建一个新实例需要使用 new 关键字，如下所示：

```
var jeremy = new Person;
```

上面这条语句将创建出 Person 对象的一个新实例 jeremy。有了这个新实例，我们就可以像下面这样利用 Person 对象的属性来检索关于 jeremy 的信息了：

```
jeremy.age
jeremy.mood
```

对象、属性、方法和实例等概念都比较抽象，为了让大家对这些概念有一个直观的认识，我在这里把虚构的 Person 对象作为例子。Person 对象在 JavaScript 语言里并不存在。我们可以利用 JavaScript 语言来创建自己的对象——术语称之为*用户定义对象*（user-defined object）。用户定义对象的创建工作是一个相当高级的话题，我们眼下还无需对它做进一步讨论。

在电视上的烹饪节目里，只要镜头一转，厨师就可以端出一盘美味的菜肴并向大家介绍说："这是我刚做好的"。JavaScript 与这种节目里的主持人颇有几分相似：它提供了一系列预先定义好的对象，而我们可以把这些对象直接用在自己的脚本里。人们把这些对象称为*内建对象*（native object）。

2.9.1 内建对象

你们其实已经见过一些 JavaScript 内建对象了。数组就是一种 JavaScript 内建对象。当我们使用 new 关键字去初始化一个数组时，其实是在创建一个 Array 对象的新实例：

```
var beatles = new Array();
```

当需要了解某个数组有多少个元素时，我们利用 Array 对象的 length 属性来获得这一信息：

```
beatles.length;
```

Array 对象只是诸多 JavaScript 内建对象中的一种。其他例子包括 Math 对象和 Date 对象，它们分别提供了许多非常有用的方法供人们与数值和日期值打交道。例如，Math 对象的 round 方法可以把十进制数值舍入为一个与之最接近的整数：

```
var num = 7.561;
var num = Math.round(num);
alert(num);
```

Date 对象可以用来存储和检索与一个特定的日期和时间有关的信息。在创建 Date 对象的新实例时，JavaScript 解释器将自动地使用当前日期和时间对它进行初始化：

```
var current_date = new Date();
```

Date 对象提供了 getDay()、getHours()、getMonth()等一系列方法，以供人们用来检索与特定日期有关的各种信息。例如，getDay()方法可以告诉我们给定日期是星期几：

```
var today = current_date.getDay();
```

在编写 JavaScript 脚本时，内建对象可以帮助我们快速、简单地完成许多任务。

2.9.2 宿主对象

除了各种 JavaScript 内建对象，我们还可以在 JavaScript 脚本里使用其他一些已经预先定义好的对象。后者不是由 JavaScript 语言本身而是由它的运行环境提供的。具体到 Web 应用，这个环境就是各种 Web 浏览器。由 Web 浏览器提供的预定义对象被称为宿主对象（host object）。

宿主对象主要包括 Form、Image 和 Element。我们可以通过这些对象获得关于某给定网页上的表单、图像和各种表单元素的信息。

虽然宿主对象很有用，但本书没有收录任何这方面的例子。因为还有一种对象可以用来获得关于某给定网页上的任何一个元素的信息，它就是 document 对象。在本书的后续内容里，我们将向大家介绍许多隶属于 document 对象的属性和方法。

2.10 小结

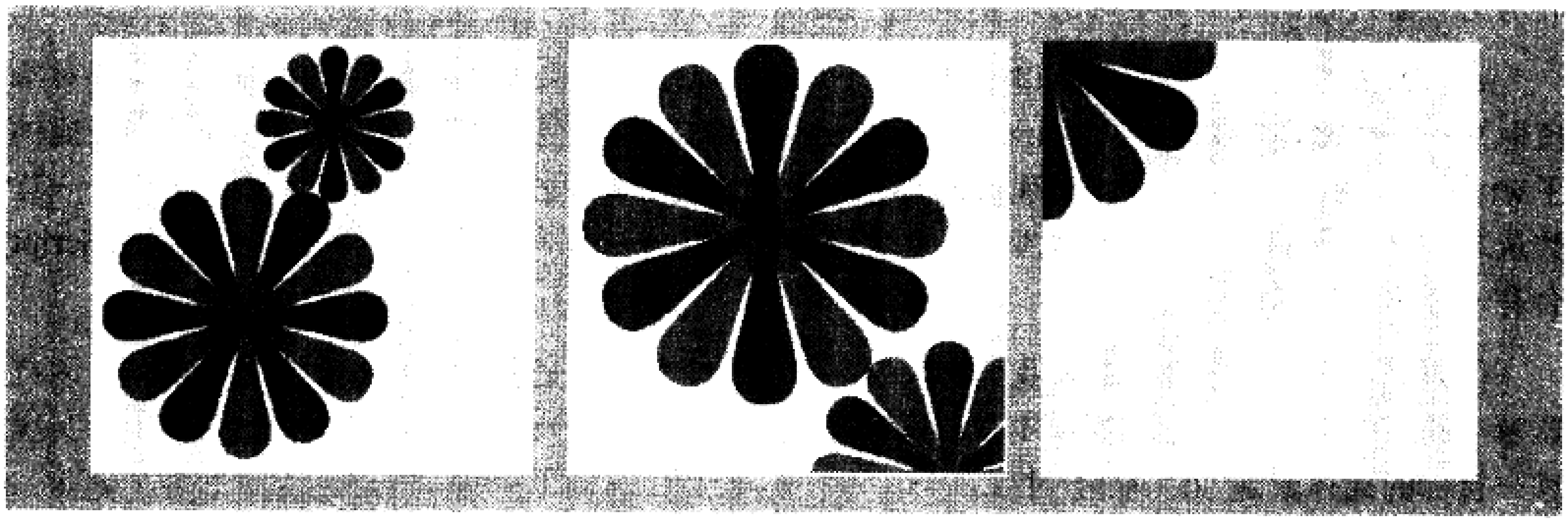
在这一章中，我们介绍了 JavaScript 语言的基本语法。在本书的后续章节中，我们还会用到这里介绍的许多概念和术语：语句、变量、数组和函数等。这些概念有的现在还不太容易理解，但我相信你们在看过它们在脚本里的实际应用后一定会加深对它们的领悟。在后面的学习里，如果需要重温这些基本概念的含义，你们随时可以返回到这一章来。

本章只对“对象”做了一个概念性的介绍。如果你们对它的理解还不够全面深入，别着急。我们将在下一章对 document 对象做进一步的探讨。我们将先向大家介绍一些与这个对象相关联的属性和方法。这些属性和方法都是由 W3C 的标准化 DOM 提供的。

在下一章中，我们将向大家介绍基于 DOM 的基本编程思路并向大家演示如何使用它的一些功能非常强大的方法。

第 3 章

DOM



本章内容

- 节点的概念
- 四个非常实用的 DOM 方法：getElementById、getElementsByTagName、getAttribute 和 setAttribute

终于要与 DOM 面对面了。能够向大家介绍 DOM 是笔者的荣幸，我非常乐于带领大家通过 DOM 的眼睛去看世界。

3.1 文档：DOM 中的“D”

DOM 是“Document Object Model”（文档对象模型）的首字母缩写。如果没有 document（文档），DOM 也就无从谈起。当创建了一个网页并把它加载到 Web 浏览器中时，DOM 就在幕后悄然而生。它将根据你编写的网页文档创建一个文档对象。

在人类语言中，“对象”这个词的含义往往不那么明确和具体，它几乎可以用来称呼任何一种客观存在的事物。但在程序设计语言中，“对象”这个词的含义非常明确和具体。

3.2 对象：DOM 中的“O”

在上一章的末尾，我们向大家展示了几个 JavaScript 对象的例子。你们应该还记得，“对象”是一种独立的数据集合。与某个特定对象相关联的变量被称为这个对象的属性；可以通过某个特

定对象去调用的函数被称为这个对象的方法。

JavaScript 语言里的对象可以分为三种类型：

- 用户定义对象（user-defined object）：由程序员自行创建的对象。本书不讨论这种对象。
- 内建对象（native object）：内建在 JavaScript 语言里的对象，如 `Array`、`Math` 和 `Date` 等。
- 宿主对象（host object）：由浏览器提供的对象。

在 JavaScript 语言的发展初期，程序员在编写 JavaScript 脚本时经常需要用到一些非常重要的宿主对象，它们当中最基础的是 `window` 对象。

`window` 对象对应着浏览器窗口本身，这个对象的属性和方法通常被统称为 BOM（浏览器对象模型）——但我觉得称之为 Window Object Model（窗口对象模型）更为贴切。BOM 向程序员提供了 `window.open` 和 `window.blur` 等方法，你们在上网冲浪时看到的各种弹出窗口和下拉菜单——其数量之多已经到了泛滥成灾的地步——几乎都是由这些方法负责创建和处理的。难怪 JavaScript 会有一个不好的名声！

值得庆幸的是，在这本书里我们不需要与 BOM 打太多的交道。我们将把注意力集中在浏览器窗口的内部而不是浏览器窗口本身。我们将着重探讨如何对网页的内容进行处理，而用来实现这一目标的载体就是 `document` 对象。

在本书的后续内容里，我们将尽可能地只讨论 `document` 对象的属性和方法。

现在，我们已经对 DOM 中的字母“D”（document，文档）和字母“O”（object，对象）做了解释，那么字母“M”又代表着什么呢？

3.3 模型：DOM 中的“M”

DOM 中的“M”代表着“Model”（模型），但说它代表着“Map”（地图）也未尝不可。模型也好，地图也罢，它们的含义都是某种事物的表现形式。就像一个模型火车代表着一列真正的火车、一张城市街道图代表着一个实际存在的城市那样，DOM 代表着被加载到浏览器窗口里的当前网页：浏览器向我们提供了当前网页的地图（或者说模型），而我们可以通过 JavaScript 去读取这张地图。

既然是地图，就必须有诸如方向、等高线和比例尺之类的记号。要想看懂和使用地图，就必须知道这些记号的含义和用途——这个道理同样适用于 DOM。要想从 DOM 获得信息，我们必须先把各种用来表示和描述一份文档的记号弄明白。

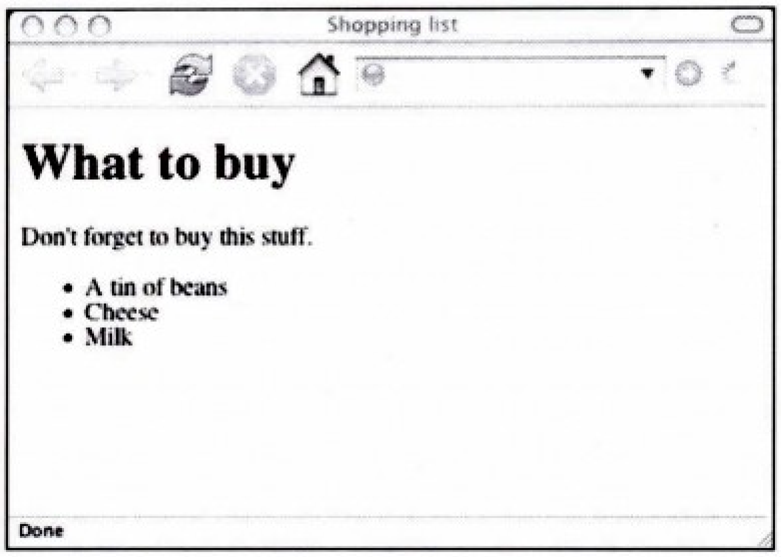
DOM 把一份文档表示为一棵树（这里所说的“树”是数学意义上的概念），这是我们理解和运用这一模型的关键。更具体地说，DOM 把文档表示为一棵家谱树。

家谱树本身又是一种模型。家谱树的典型用法是表示一个人类家族的谱系并使用 parent（父）、child（子）、sibling（兄弟）等记号来表明家族成员之间的关系。家谱树可以把一些相当复杂的

关系简明地表示出来：一位特定的家族成员既是某些成员的父辈，又是另一位成员的子辈，同时还是另一位成员的兄弟。

类似于人类家族谱系的情况，家谱树模型也非常适合用来表示一份用(X)HTML 语言编写出来的文档。

请看下面这份非常基本的网页，它的内容是一份购物清单。




```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html;
➥ charset=utf-8" />
    <title>Shopping list</title>
  </head>
  <body>
    <h1>What to buy</h1>
    <p title="a gentle reminder">Don't forget to buy this stuff.</p>
    <ul id="purchases">
      <li>A tin of beans</li>
      <li>Cheese</li>
      <li>Milk</li>
    </ul>
  </body>
</html>
```

这份文档可以用图 3-1 中的模型来表示。

我们来分析一下这个网页的结构。这种分析不仅可以让我们了解它是由哪些元素构成的，还可以让我们了解为什么图 3-1 中的模型可以如此完美地把它表示出来。在对 Doctype 做出声明后，这份文档首先打开了一个<html>标签，而这个网页里的所有其他元素都包含在这个元素里。因为所有其他的元素都包含在其内部，所以这个<html>标签既没有父辈，也没有兄弟。如果这是一棵真正的树的话，这个<html>标签显然就是树根。

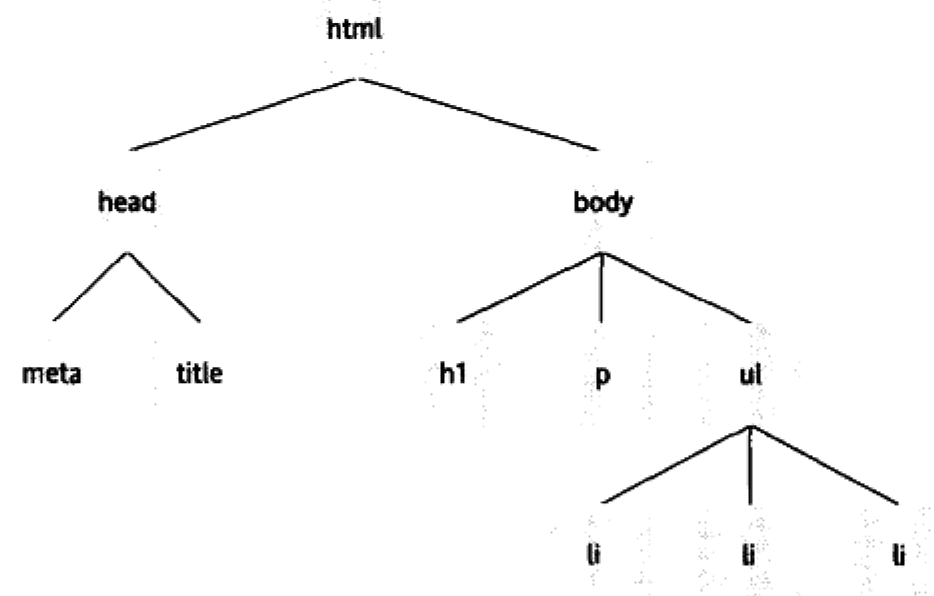


图 3-1 把网页中的元素表示为一棵家谱树

这正是图 3-1 中的家谱树以 html 为根元素的原因。毫无疑问，html 元素完全可以代表整个文档。

如果在这份文档里更深入一层，我们将发现<head>和<body>两个分枝。它们存在于同一层次且互不包含，所以它们是兄弟关系。它们有着共同的父元素<html>，但又各有各的子元素，所以它们本身又是其他一些元素的父元素。

<head>元素有两个子元素：<meta>和<title>（这两个元素是兄弟关系）。<body>元素有三个子元素：<h1>、<p>和<ul>（这三个元素是兄弟关系）。如果继续深入下去，我们将发现<ul>也是一个父元素。它有三个子元素，它们都是<li>元素。

利用这种简单的家谱关系记号，我们可以把各元素之间的关系简明清晰地表达出来。

例如，<h1>和<ul>之间是什么关系？答案是它们是兄弟关系。

那么<body>和<ul>之间又是什么关系？<body>是<ul>的父元素，<ul>是<body>的一个子元素。

如果把各种文档元素想像成一棵家谱树上的各个节点的话，我们就可以用同样的记号来描述 DOM。不过，与使用“家谱树”这个术语相比，把一份文档称为一棵“节点树”更准确。

3.3.1 节点

节点（node）这个名词来自网络理论，它代表着网络中的一个连接点。网络是由节点构成的集合。

在现实世界里，一切事物都由原子构成。原子是现实世界的节点。但原子本身还可以进一步分解为更细小的亚原子微粒。这些亚原子微粒同样是节点。

DOM 也是同样的情况。文档也是由节点构成的集合，只不过此时的节点是文档树上的树枝和树叶而已。

在 DOM 里存在着许多不同类型的节点。就像原子包含着亚原子微粒那样，有些 DOM 节点类型还包含着其他类型的节点。

1. 元素节点

DOM 的原子是元素节点（element node）。

在描述刚才那份“购物清单”文档时，我们使用了诸如<body>、<p>和<ul>之类的元素。如果把 Web 上的文档比作一座大厦，元素就是建造这座大厦的砖块，这些元素在文档中的布局形成了文档的结构。

各种标签提供了元素的名字。文本段落元素的名字是“p”，无序清单元素的名字是“ul”，列表项元素的名字是“li”。

元素可以包含其他的元素。在我们的“购物清单”文档里，所有的列表项元素都包含在一个无序清单元素的内部。事实上，没有被包含在其他元素里的唯一元素是<html>元素。它是我们的节点树的根元素。

2. 文本节点

元素只是不同节点类型中的一种。如果一份文档完全由一些空白元素构成，它将有一个结构，但这份文档本身将不会包含什么内容。在网上，内容决定着一切，没有内容的文档是没有任何价值的，而绝大多数内容都是由文本提供的。

在“购物清单”例子里，<p>元素包含着文本“Don’t forget to buy this stuff.”。它是一个文本节点（text node）。

在 XHTML 文档里，文本节点总是被包含在元素节点的内部。但并非所有的元素节点都包含有文本节点。在“购物清单”文档里，<ul>元素没有直接包含任何文本节点——它包含着其他的元素节点（一些<li>元素），后者包含着文本节点。

3. 属性节点

还存在着其他一些节点类型。例如，注释就是另外一种节点类型。但我们这里还想向大家再多介绍一种节点类型。

元素都或多或少地有一些属性，属性的作用是对元素做出更具体的描述。例如，几乎所有的元素都有一个 title 属性，而我们可以利用这个属性对包含在元素里的东西做出准确的描述：

```
<p title="a gentle reminder">Don’t forget to buy this stuff.</p>
```

在 DOM 中，title="a gentle reminder"是一个属性节点（attribute node），如图 3-2 所示。因为属性总是被放在起始标签里，所以属性节点总是被包含在元素节点当中。

并非所有的元素都包含着属性，但所有的属性都会被包含在元素里。



图 3-2 一个元素节点，它包含着一个属性节点和一个文本节点

在前面的“购物清单”示例文档里，我们可以清楚地看到那个无序清单元素（<ul>）有个 id 属性。如果曾经使用过 CSS，你们对 id 和 class 之类的属性应该不会感到陌生。不过，为了照顾那些对 CSS 还不太熟悉的读者，我们下面将简要地重温几个最基本的 CSS 概念。

4. CSS：层叠样式表

DOM 并不是人们与网页的结构打交道的唯一手段。我们还可以通过 CSS（层叠样式表）告诉浏览器应该如何显示一份文档的内容。

类似于 JavaScript 脚本，对样式的声明既可以嵌在文档的<head>部分（这需要用到<style>标签），也可以放在另外一个样式表文件里。利用 CSS 对某个元素的样式做出声明的语法与 JavaScript 函数的定义语法很相似：

```
selector {
 property: value;
}
```

在样式声明里，我们可以对浏览器在显示各有关元素时使用的颜色、字体和字号做出定义，如下所示：

```
p {
  color: yellow;
  font-family: "arial", sans-serif;
  font-size: 1.2em;
}
```

继承（inheritance）是 CSS 技术中的一项强大功能。类似于 DOM，CSS 也把文档的内容视为一棵节点树。节点树上的各个元素将继承其父元素的样式属性。

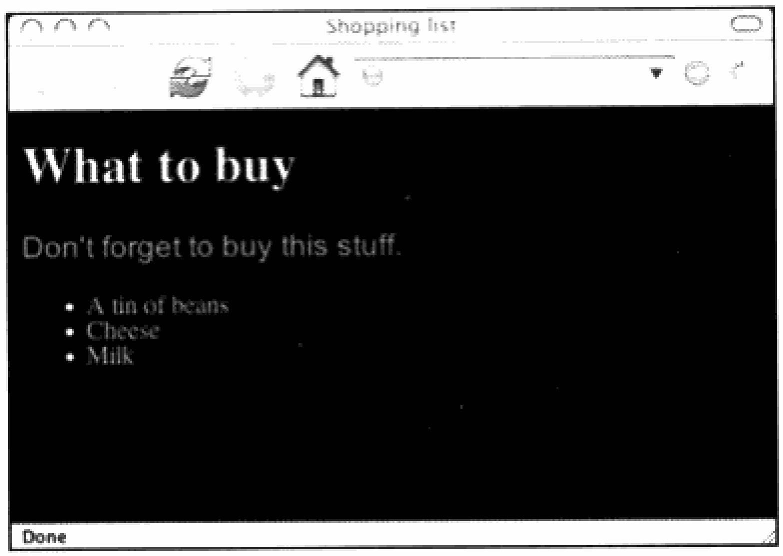
例如，如果我们为 body 元素定义了一些颜色或字体，包含在 body 元素里的所有元素都将自

动获得——也就是继承，那些样式：

```
body {
  color: white;
  background-color: black;
}
```

这些颜色将不仅作用于那些被直接包含在<body>标签里的内容，还将作用于那些嵌套在 body 元素内部的所有元素。

这里是把刚才定义的样式应用在“购物清单”示例文档上而得到的网页显示效果。



在某些场合，当把样式应用于一份文档时，我们其实只想让那些样式只作用于某个特定的元素。例如，我们只想让某一段文本变成某种特殊的颜色和字体，但不想让其他段落受到影响。为了获得如此精细的控制，我们将需要在文档里插入一些能够把这段文本与其他段落区别开来的特殊标志。

为了把某一个或某几个元素与其他元素区别开来，我们需要使用 class 属性或 id 属性之一。

- class属性

所有的元素都有 class 属性，不同的元素可以有同样的 class 属性值。如下所示：

```
<p class="special">This paragraph has the special class</p>
<h2 class="special">So does this headline</h2>
```

在样式表里，我们可以像下面这样为 class 属性值相同的所有元素定义一种共享的样式：

```
.special {
  font-style: italic;
}
```

我们还可以像下面这样利用 class 属性为一种特定类型的元素定义一种独享的样式：

```
h2.special {
  text-transform: uppercase;
}
```

- id属性

id 属性的用途是给网页里的某个元素加上一个独一无二的标识符，如下所示：

```
<ul id="purchases">
```

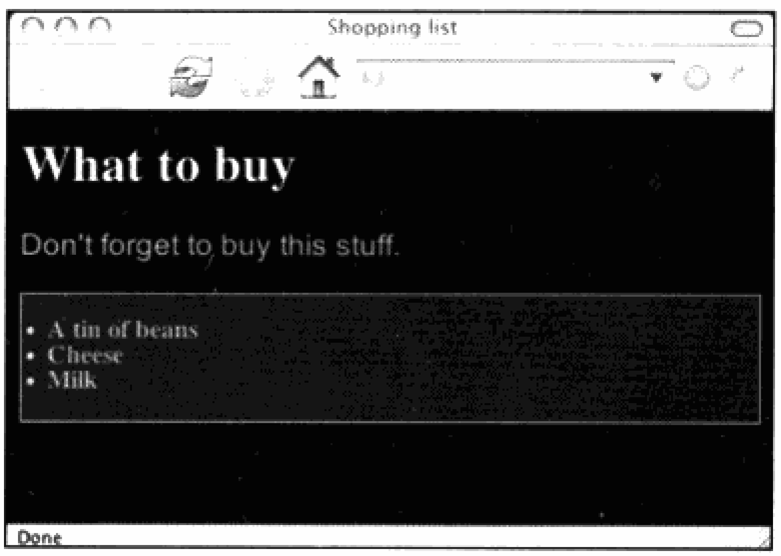
在样式表里，我们可以像下面这样为有着特定 id 属性值的元素定义一种独享的样式：

```
#purchases {
  border: 1px solid white;
  background-color: #333;
  color: #ccc;
  padding: 1em;
}
```

每个元素只能有一个 id 属性值，不同的元素必须有不同的 id 属性值。不过，我们可以在样式表里像下面这样，利用 id 属性为包含在某给定元素里的其他元素定义样式，而如此定义出来的样式将只作用于包含在这个给定元素里的有关元素：

```
#purchases li {
  font-weight: bold;
}
```

这里是把刚才利用 id 属性定义的样式应用在“购物清单”示例文档上而得到的网页显示效果。



id 属性就像是一个挂钩，它一头连着文档里的某个元素，另一头连着 CSS 样式表里的某个样式。DOM 也可以使用这种挂钩，事实上，有不少 DOM 操作必须借助于这种挂钩才能完成。

3.3.2 getElementById()方法

DOM 提供了一个名为 getElementById()的方法，这个方法将返回一个与那个有着给定 id 属性值的元素节点相对应的对象。请注意，JavaScript 语言区分字母的大小写情况，所以大家在写出“getElementById”时千万不要把大小写弄错了。如果把它错写成“GetElementById”或“getElementbyid”，你将无法得到你真正想要的东西。

这个方法是与 document 对象相关联的函数。在脚本代码里，函数名的后面必须跟有一组圆括号，这组圆括号包含着函数的参数。getElementById()方法只有一个参数：你想获得的那个元素的 id 属性值，这个 id 值必须放在单引号或双引号里。

document.getElementById(id)

下面是一个例子：

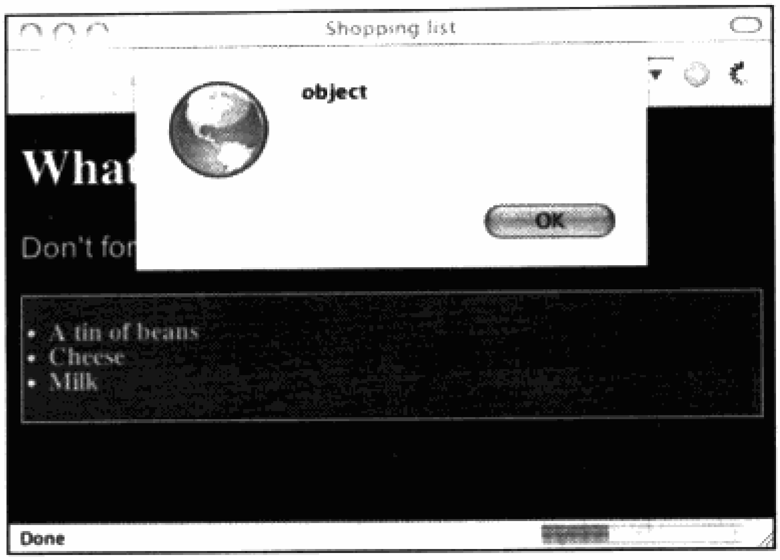
```
document.getElementById("purchases")
```

这个调用将返回一个对象，这个对象对应着 document 对象里的一个独一无二的元素，那个元素的 HTML id 属性值是 purchases。再说一遍，getElementById()方法将返回一个对象。你们可以用 typeof 操作符来验证这一点。typeof 操作符可以告诉我们它的操作数是不是一个字符串、数值、函数、布尔值或对象。

下面是把一些 JavaScript 语句插入到前面给出的“购物清单”文档之后得到的一份代码清单，新增的代码（黑体字部分）出现在</body>结束标签之前。顺便说一句，我本人并不赞成把 JavaScript 代码直接嵌入一份文档的做法，但它不失为一种简便快捷的测试手段：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html;
➥ charset=utf-8" />
    <title>Shopping list</title>
  </head>
  <body>
    <h1>What to buy</h1>
    <p title="a gentle reminder">Don't forget to buy this stuff.</p>
    <ul id="purchases">
      <li>A tin of beans</li>
      <li>Cheese</li>
      <li>Milk</li>
    </ul>
<script type="text/javascript">
alert(typeof document.getElementById("purchases"));
</script>
  </body>
</html>
```

把上面这些代码保存为一个 XHTML 文件。当在你们的 Web 浏览器里加载这个 XHTML 文件时，屏幕上将弹出一个如下所示的 alter 对话框，它向你们报告 document.getElementById("purchases")的类型——它是一个对象。不仅如此，如果用上述办法去检查其他元素节点的类型，你们也会看到类似的 alert 对话框。



事实上，文档中的每一个元素都对应着一个对象。利用 DOM 提供的方法，我们可以把与这些元素相对应的任何一个对象筛选出来。

一般来说，我们用不着为文档里的每一个元素都分别定义一个独一无二的 id 值；那也太小题大做了。要知道，即使某个元素没有独一无二的 id 值，我们也可以利用 DOM 提供的另一个方法把与之对应的对象准确无误地筛选出来。

3.3.3 getElementsByTagName()方法

getElementsByTagName()方法将返回一个对象数组，每个对象分别对应着文档里有着给定标签的一个元素。类似于 getElementById()，这个方法也是只有一个参数的函数，它的参数是(X)HTML 标签的名字：

```
element.getElementsByTagName(tag)
```

它与 getElementById()方法有许多相似之处，但有一点要特别提醒大家注意：getElementsByTagName()方法返回的是一个数组；你们在编写脚本时千万注意不要把这两个方法弄混了。

下面是一个例子：

```
document.getElementsByTagName("li")
```

这个调用将返回一个对象数组，每个对象分别对应着 document 对象中的一个列表项（li）元素。与任何其他的数组一样，我们可以利用 length 属性查出这个数组里的元素个数。

首先，在上一小节给出的 XHTML 示例文档里把<script>标签中的 alter 语句替换为下面这条语句：

```
alert(document.getElementsByTagName("li").length);
```

然后，用浏览器里重新加载那个 XHTML 文件，你们就会看到这份示例文档里的列表项元素的个数：3。这个数组里的每个元素都是一个对象。可以通过利用一个循环语句和 typeof 操作符去遍历这个数组的办法来验证这一点。例如，你们可以试试下面这个 for 循环：

```
for (var i=0; i < document.getElementsByTagName("li").length; i++) {
  alert(typeof document.getElementsByTagName("li")[i]);
}
```

请注意，即使在整个文档里只有一个元素有着给定的标签名，getElementsByTagName()方法也将返回一个数组。此时，那个数组的长度是 1。

你们或许早就觉得用键盘反复敲入 document.getElementsByTagName("li")是件很麻烦的事情，而这些长长的字符串会让代码变得越来越难以阅读。有个简单的办法可以减少不必要的打字量并改善代码的可读性：只要把 document.getElementsByTagName("li")赋值给一个变量即可。

请在上一小节给出的 XHTML 示例文档里把<script>标签中的 alter 语句替换为下面这些语句：

```
var items = document.getElementsByTagName("li");
for (var i=0; i < items.length; i++) {
  alert(typeof items[i]);
}
```

现在，当用浏览器里重新加载那个 XHTML 文件时，你们将看到三个 alert 对话框，显示在这三个 alert 对话框里的消息都是“object”。

getElementsByTagName()方法允许我们把一个通配符当为它的参数，而这意味着文档里的每个元素都将在这个函数所返回的数组里占有一席之地。通配符（星号字符“*”）必须放在引号里，这是为了让通配符与乘法操作符有所区别。如果你想知道某份文档里总共有多少个元素节点，像下面这样以通配符为参数去调用 getElementsByTagName()方法是最简便的办法：

```
alert(document.getElementsByTagName("*").length);
```

我们还可以把 getElementById()和 getElementsByTagName()方法结合起来运用。例如，我们刚才给出的几个例子都是通过 document 对象调用 getElementsByTagName()方法的，如果只想知道其 id 属性值是 purchase 的元素包含着多少个列表项（li）的话，你就必须通过一个更具体的对象去调用这个方法，如下所示：

```
var shopping = document.getElementById("purchases");
var items = shopping.getElementsByTagName("*");
```

在这两条语句执行完毕后，items 数组将只包含其 id 属性值是 purchase 的无序清单里的元素。具体到我们这个例子，items 数组的长度刚好与这份文档里的列表项元素的总数相等：

```
alert (items.length);
```

如果还需要更多的证据，下面这些语句将证明 items 数组里的每个值确实是一个对象：

```
for (var i=0; i < items.length; i++) {
  alert(typeof items[i]);
}
```

3.4 趁热打铁

在看过那么多显示着单词“object”的 alert 对话框后，你们很可能不愿意再看到它。如果真是如此，我的目的也就达到了——我想通过这些 alert 对话框强调这样一个事实：文档中的每个元素节点都是一个对象。我现在要告诉大家的是，这些对象中的每个都为我们准备了一系列非常有用的方法，而这一切当然都要归功于 DOM。利用这些已经预先定义好的方法，我们不仅可以检索出关于文档里的任何一个对象的信息，甚至还可以改变某些元素的属性。

下面是对本章此前学习内容的一个简要总结：

- 一份文档就是一棵节点树。
- 节点分为不同的类型：元素节点、属性节点和文本节点等。
- getElementById()方法将返回一个对象，该对象对应着文档里的一个特定的元素节点。
- getElementsByTagName()方法将返回一个对象数组，它们分别对应着文档里的一个特定的元素节点。
- 这些节点中的每个都是一个对象。

接下来，我们将向大家介绍几个与这些对象相关联的属性和方法。

3.4.1 getAttribute()方法

至此，我们已经向大家介绍了两种检索特定元素节点的办法：一种是使用 getElementById()方法，另一种是使用 getElementsByTagName()方法。在找到那个元素后，我们就可以利用 getAttribute()方法把它的各种属性的值查询出来。

getAttribute()方法是一个函数。它只有一个参数——你打算查询的属性的名字：

object.getAttribute(attribute)

不过，getAttribute()方法不能通过 document 对象调用，这与我们此前介绍过的其他方法不同。我们只能通过一个元素节点对象调用它。

例如，你可以把它与 getElementsByTagName()方法结合起来，去查询每个<p>元素的 title 属性，如下所示：

```
var paras = document.getElementsByTagName("p");
for (var i=0; i < paras.length; i++ ) {
  alert(paras[i].getAttribute("title"));
}
```

如果把上面这段代码插入到前面给出的“购物清单”示例文档的末尾，并在 Web 浏览器里重新加载这个页面，屏幕上将弹出一个显示着文本消息“a gentle reminder”的 alter 对话框。

在“购物清单”文档里只有一个带有 title 属性的<p>元素。假如这份文档还有一个或更多

个不带 title 属性的<p>元素，则相应的 getAttribute("title")调用将返回 null。null 是 JavaScript 语言中的空值，其含义是“你说的这个东西不存在”。如果你们想亲自验证一下这件事，请先把下面这段文本插入到“购物清单”文档中的现有文本段落之后：

```
<p>This is just a test</p>
```

然后重新加载这个页面。这一次，你们将看到两个 alter 对话框，而第二个对话框将是一片空白或者是只显示着单词“null”——具体情况要取决于你的 Web 浏览器将如何显示 null 值。

可以修改我们的脚本，让它只在 title 属性存在时才弹出一条消息。我们将增加一条 if 语句来检查 getAttribute()方法的返回值是不是 null。趁着这个机会，我们还增加了几个变量以提高脚本的可读性：

```
var paras = document.getElementsByTagName("p");
for (var i=0; i< paras.length; i++) {
  var title_text = paras[i].getAttribute("title");
  if (title_text != null) {
    alert(title_text);
  }
}
```

现在，如果重新加载这个页面，你们将只会看到一个显示着“a gentle reminder”消息的 alter 对话框，如下所示。



我们甚至可以把这段代码缩得更短一些。当检查某项数据是否是 null 值时，我们其实是在检查它是否存在。这种检查可以简化为直接把被检查的数据用做 if 语句的条件。if (something) 与 if (something != null)完全等价，但前者显然更为简明。此时，如果 something 存在，则 if 语句的条件将为真；如果 something 不存在，则 if 语句的条件将为假。

具体到这个例子，只要我们把 if (title_text != null)替换为 if (title_text)，我们就可以得到更简明的代码。此外，为了进一步增加代码的可读性，我们还可以趁此机会把 alter 语句与 if 语句写在同一行上，这可以让它们更接近于我们日常生活中的英语句子：

```
var paras = document.getElementsByTagName("p");
for (var i=0; i< paras.length; i++) {
  var title_text = paras[i].getAttribute("title");
  if (title_text) alert(title_text);
}
```

3.4.2 setAttribute()方法

我们此前介绍给大家的所有方法都只能用来检索信息。setAttribute()方法与它们有一个本质上的区别：它允许我们对属性节点的值做出修改。

类似于 getAttribute()方法，setAttribute()方法也是一个只能通过元素节点对象调用的函数，但 setAttribute()方法需要我们向它传递两个参数：

object.setAttribute(attribute,value)

在下面的例子里，第一条语句将把 id 属性值是 purchase 的元素检索出来，第二条语句将把这个元素的 title 属性值设置为 a list of goods：

```
var shopping = document.getElementById("purchases");
shopping.setAttribute("title","a list of goods");
```

我们可以利用 getAttribute()方法来证明这个元素的 title 属性值确实发生了变化：

```
var shopping = document.getElementById("purchases");
alert(shopping.getAttribute("title"));
shopping.setAttribute("title","a list of goods");
alert(shopping.getAttribute("title"));
```

上面这些语句将在屏幕上弹出两个 alert 对话框：第一个 alter 对话框出现在 setAttribute()方法被调用之前，它将是一片空白或显示着单词“null”；第二个出现在 title 属性值被设置之后，它将显示着“a list of goods”消息。

在上例中，我们设置了一个现有节点的 title 属性，但这个属性原先并不存在。这意味着我们发出的 setAttribute()调用实际完成了两项操作：先把这个属性创建出来，然后再对其值进行设置。如果我们把 setAttribute()方法用在元素节点的某个现有属性上，这个属性的当前值将被覆盖。

在“购物清单”示例文档里，<p>元素已经有了一个 title 属性，这个属性的值是 a gentle reminder。我们可以用 setAttribute()方法来改变它的当前值：

```
var paras = document.getElementsByTagName("p");
for (var i=0; i< paras.length; i++) {
  var title_text = paras[i].getAttribute("title");
  if (title_text) {
    paras[i].setAttribute("title","brand new title text");
    alert(paras[i].getAttribute("title"));
  }
}
```

上面这段代码将先从文档里把已经带有 title 属性的<p>元素全部检索出来，然后把它们的 title 属性值全部修改为 brand new title text。具体到“购物清单”文档，属性值 a gentle reminder 将被覆盖。

这里有一个非常值得关注的细节：通过 setAttribute()方法对文档做出的修改，将使得文档在浏览器窗口里的显示效果和/或行为动作发生相应的变化，但我们在通过浏览器的 view source（查看源代码）选项去查看文档的源代码时看到的仍将是原来的属性值——也就是说，setAttribute()方法做出的修改不会反映在文档本身的源代码里。这种“表里不一”的现象源自 DOM 的工作模式：先加载文档的静态内容、再以动态方式对它们进行刷新，动态刷新不影响文档的静态内容。这正是 DOM 的真正威力和诱人之处：对页面内容的刷新不需要最终用户在他们的浏览器里执行页面刷新操作就可以实现。

3.5 小结

在这一章里，我们向大家介绍了 DOM 提供的四个方法：

- getElementById()方法
- getElementsByTagName()方法
- getAttribute()方法
- setAttribute()方法

这四个方法是将要编写的许多 DOM 脚本的基石。

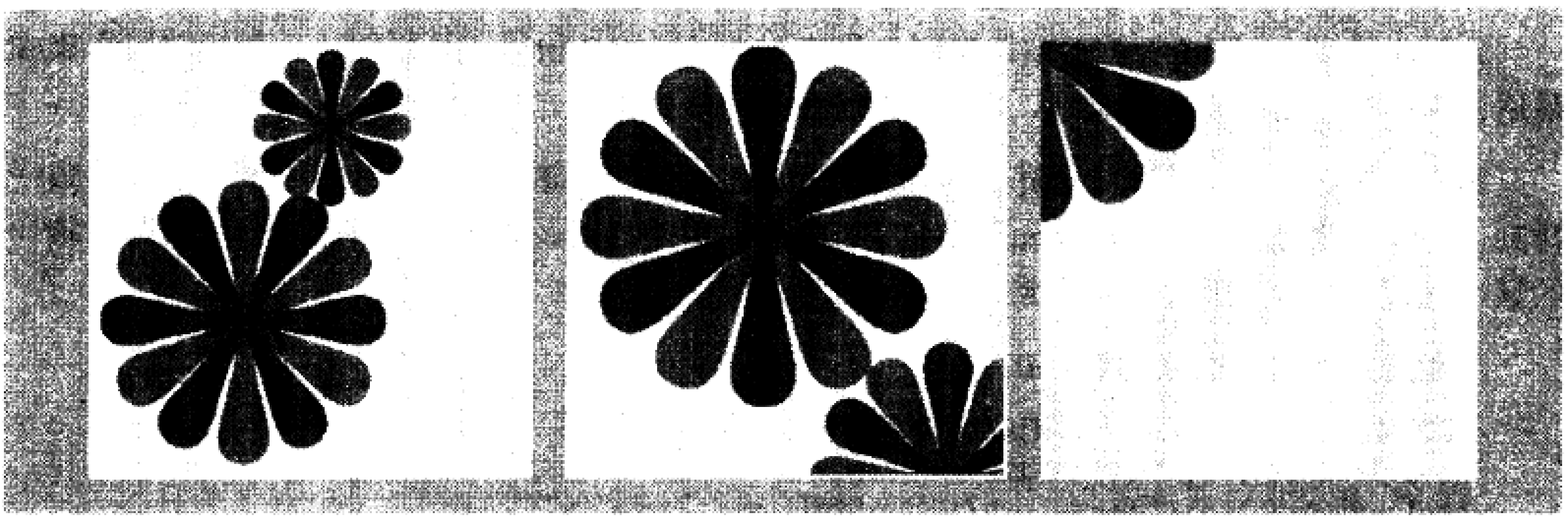
DOM 还提供了许多其他的属性和方法，如 nodeName、nodeValue、chileNodes、nextSibling 和 parentNode 等，但现在还不是它们出场的时候——我打算在本书后面的有关章节中选择一些适当的时机把它们依次介绍给大家。我在这里列出它们的目的只是为了激起大家的学习兴趣。

本章内容比较偏重于理论。在看过那么多的 alter 对话框之后，相信大家都迫不及待地想通过其他一些东西去进一步了解和测试 DOM，而我也正想通过一个案例来进一步展示 DOM 的强大威力。

在下一章中，我将带领大家利用本章介绍的四个 DOM 方法去创建一个基于 JavaScript 的图片库。

第 4 章

案例研究：JavaScript 美术馆



现在，是时候利用 DOM 去完成一些简单的任务了。在这一章中，我将带领大家用 JavaScript 和 DOM 去建立一个图片库，并将其称为“JavaScript 美术馆”。我们将按照以下步骤来完成这一案例：

- 编写一份优秀的标记语言文档[1]。
- 编写一个 JavaScript 函数以显示用户想要查看的图片。
- 在标记语言文档里触发一个调用 JavaScript 图片显示的函数。
- 对这个 JavaScript 函数的功能进行扩展。这需要用到几个新的 DOM 方法。

把一些图片发布到网上的办法很多。最容易想到的办法是把所有的图片都放到同一个网页里。不过，如果你打算发布的图片比较多的话，这个页面很快就会因为变得过于巨大而失去吸引力。要知道，虽然这种网页本身的 HTML 代码不会多到哪儿去，但算上那些图片可就不一样了。我们必须面对的现实是：数据量越大，网页的下载时间就越长，但愿意等待很长时间去下载一个网页的人却没有几个。

因此，为每张图片分别创建一个网页的解决方案显然更值得考虑。你的图片库将不再是一个体积庞大、难以下载的网页，而是许多尺寸合理、便于下载和浏览的页面。不过，这一解决方案并非尽善尽美：首先，为每张图片分别制作一个网页需要花费不少的时间和精力；其次，需要在

1. 为简明起见，本章将以“HTML 文档”或“XHTML 文档”来称呼这类文档，但 DOM 允许我们使用任何一种标准化的标记语言来编写它们。——译者注

每个网页上提供一些链接，来给出当前图片在整个图片库里的位置以方便人们从当前图片转到其他的图片。

如果你想两全其美，利用 JavaScript 来创建图片库将是最佳的选择：把整个图片库的浏览链接集中安排在你的图片库主页里，只在用户点击了这个主页里的某个图片链接时才把相应的图片传送给他。

4.1　编写标记语言文档

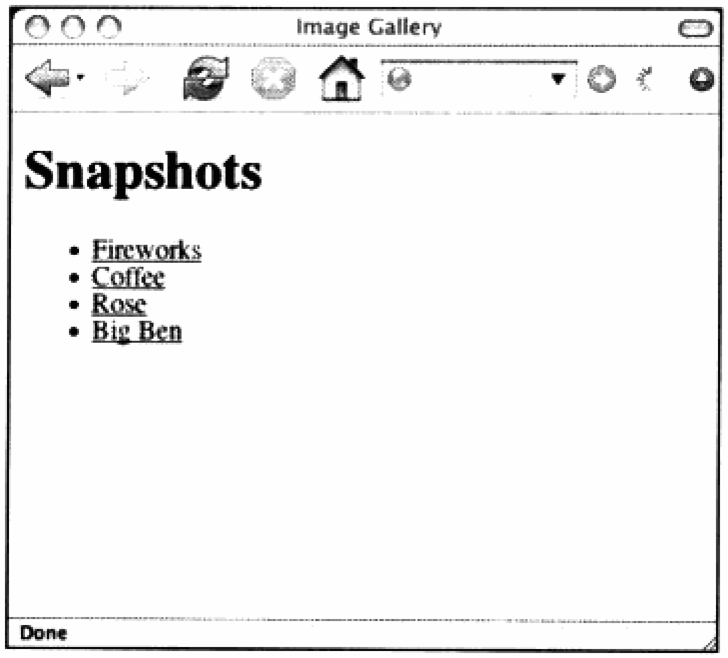
为了完成“JavaScript 美术馆”案例，我特意用数码相机拍摄了几张照片，并把它们修整成最适合于用浏览器来查看的尺寸，即 400 像素宽×300 像素高。

第一项工作是为这些图片创建一个链接清单。因为我没打算让这些图片按照某种特殊的顺序排列，所以将使用一个无序清单元素（<ul>）来列出那些链接。如果想让自己的图片按顺序排列的话，应该使用一个排序清单元素（<ol>）去组织图片链接。

下面就是我编写出来的 HTML 文档：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Image Gallery</title>
</head>
<body>
  <h1>Snapshots</h1>
  <ul>
    <li>
      <a href="images/fireworks.jpg" title="A fireworks display">
➥ Fireworks</a>
    </li>
    <li>
      <a href="images/coffee.jpg" title="A cup of black coffee">
➥ Coffee</a>
    </li>
    <li>
      <a href="images/rose.jpg" title="A red, red rose">Rose</a>
    </li>
    <li>
      <a href="images/bigben.jpg" title="The famous clock">
➥ Big Ben</a>
    </li>
  </ul>
</body>
</html>
```

我将把这份文档保存为 gallery.html 文件，并把图片集中保存在子目录 images 里。我的 images 子目录和 gallery.html 文件位于同一个子目录下。在 gallery.html 文件里，无序清单元素中的每个链接分别指向不同的图片。在浏览器窗口里点击某个链接就可以转到相应的图片，但从图片重新返回到链接清单目前还必须借助于浏览器的 Back（后退）按钮。下面是这个基本的链接清单在浏览器窗口里的显示效果。



这是一个相当令人满意的网页，但它的默认行为还不太理想。下面是我觉得还需要改进的几个地方：

- 当点击某个链接时，我希望能留在这个网页而不是转到另一个窗口。
- 当点击某个链接时，我希望能在这个网页上同时看到那张图片以及原有的图片清单。

下面是我为了实现上述希望而需要完成的几项改进：

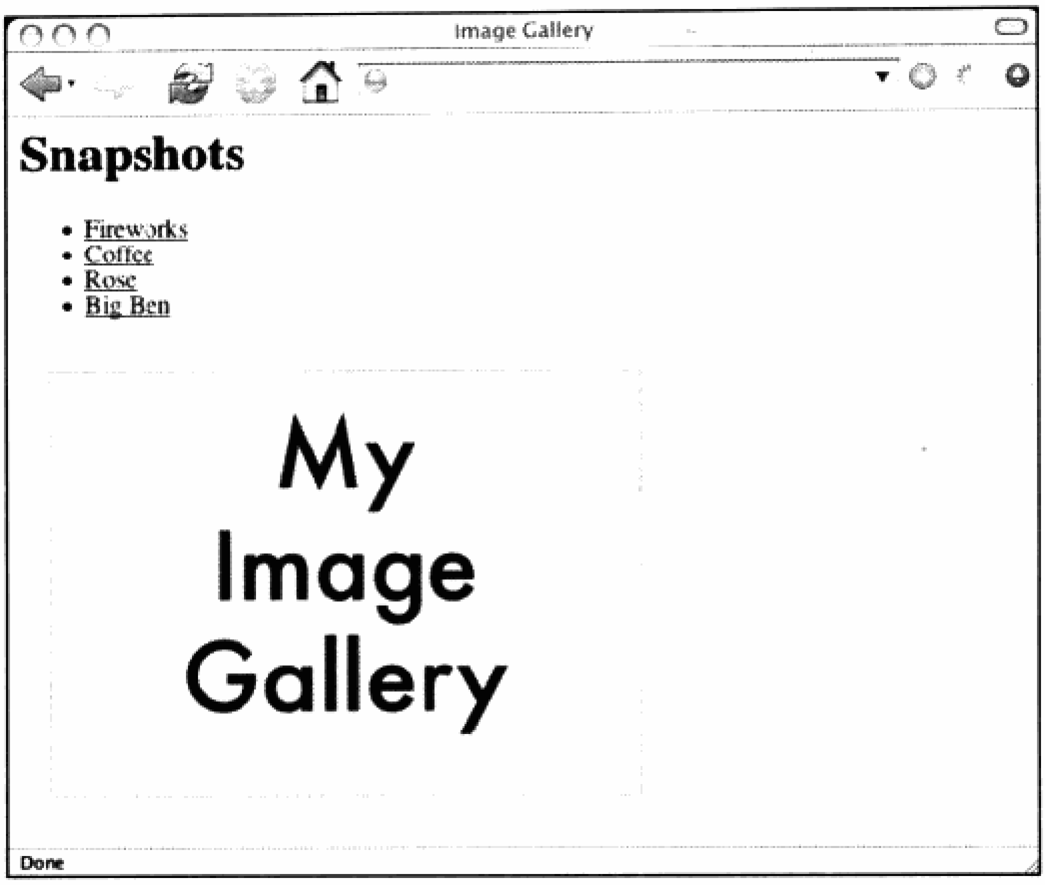
- 通过增加一个“占位符”图片的办法在这个主页上为图片预留一个浏览区域。
- 在点击某个链接时，将拦截这个网页的默认行为。
- 在点击某个链接时，将把“占位符”图片替换为与那个链接相对应的图片。

我们先来解决“占位符”图片的问题。我选用了一个类似于名片的图片，你们可以根据个人喜好来决定选用的图片，即使选用一个空白图片也没问题。

把下面这些代码插入到图片名单的末尾：

```
<img id="placeholder" src="images/placeholder.gif"
➥ alt="my image gallery" />
```

我对这个图片的 id 属性进行了设置，这将使我可以通过一个外部的样式表对图片的显示位置和显示效果加以控制。例如，我可以让这个图片出现在链接清单的旁边而不是它的下方。我还可以在自己的 JavaScript 代码里使用这个 id 值。下面是这个页面在增加了“占位符”图片后的显示效果。



现在，我的 HTML 文档已经准备好了。接下来的工作是编写一些必要的 JavaScript 代码。

4.2 编写 JavaScript 函数

为了把“占位符”图片替换为想要查看的图片，需要改变它的 src 属性。setAttribute()方法是完成这项工作的最佳选择，而我将利用这个方法写一个函数。那个函数只有一个参数，即我想查看的那张图片的链接；它将通过改变“占位符”图片的 src 属性的办法将其替换为我想查看的那张图片。

首先，需要给函数起一个好的名字。我想让它既能提醒我这个函数的用途，又足够简明扼要。我决定把这个函数命名为 showPic()。还需要给这个函数的参数起一个名字，我决定把它命名为 whichpic：

```
function showPic(whichpic)
```

whichpic 代表着一个元素节点；具体地说，那将是一个指向某个图片的<a>元素。我需要知道那个图片的文件路径，这个路径可以通过在 whichpic 元素上调用 getAttribute()方法的办法查出来——只要把"href"作为参数传递给 getAttribute()方法，就可以知道那个图片的文件路径了：

```
whichpic.getAttribute("href")
```

我将把这个路径存入一个变量以便在后面的语句里使用它。我决定把这个变量命名为 source：

```
var source = whichpic.getAttribute("href");
```

接下来，还需要把“占位符”图片检索出来，这种事对 getElementById()方法来说不过是小菜一碟：

```
document.getElementById("placeholder")
```

我不想重复敲入“document. getElementById("placerholder")”这么长的字符串，所以将把这个元素赋值给一个变量并将其命名为 placerholder：

```
var placeholder = document.getElementById("placeholder");
```

现在，已经声明并赋值了两个变量：source 和 placerholder。它们可以让我的脚本简明易读。

我将使用 setAttribute()方法对 placerholder 元素的 src 属性进行刷新。还记得吗，这个方法有两个参数：一个是打算对之进行设置的属性，另一个是这个属性的新属性值。具体到这个例子，因为我想对 src 属性进行设置，所以第一个参数是"src"；至于第二个参数，也就是 src 属性的新属性值，我已经把它保存在 source 变量里了：

```
placeholder.setAttribute("src",source);
```

这显然要比下面这么冗长的代码更容易阅读和理解：

```
document.getElementById("placeholder").setAttribute("src",
➥ whichpic.getAttribute("href"));
```

4.2.1 DOM 之前的解决方案

其实，不使用 setAttribute()方法也可以改变某个图片的 src 属性。

setAttribute()方法是“第 1 级 DOM”（DOM Level 1）的组成部分之一，这个方法可以对任意元素节点的任意属性进行设置。在“第 1 级 DOM”出现之前，程序员只能通过另外一种办法对一部分元素的属性进行设置，这个办法至少在目前还可以用来改变某些属性的值。

例如，如果想改变某个输入元素的 value 属性，可以使用如下所示的办法：

```
element.value = "the new value"
```

这与下面这条语句的效果是等价的：

```
element.setAttribute("value","the new value");
```

类似的办法也可以用来改变图片的 src 属性。例如，在我的图片库脚本里，完全可以用下面这条语句来代替 setAttribute()调用：

```
placeholder.src = source;
```

我个人更喜欢使用 setAttribute()方法。这起码可以让我不必费心去记忆哪些元素的哪些属性可以用哪些 DOM 之前的方法去设置。虽然用那些老办法可以毫无问题地对文档里的图片、表单和其他一些元素的属性进行设置，但 setAttribute()方法的优势在于它可以对文档中的任何一个元素的任何一个属性做出修改。

"第 1 级 DOM"的另一个优势是可移植性更好。那些老方法只适用于 Web 文档，DOM 则适用于任何一种标记语言。虽然这种差异对我们这个例子没有影响，但我希望大家能够牢牢记住这一点：DOM 是一种适用于多种环境和多种程序设计语言的通用型 API。如果想把从本书学到的 DOM 技巧运用在 Web 浏览器以外的应用环境里，严格遵守"第 1 级 DOM"能够让你们避免与兼容性有关的任何问题。

4.2.2 showPic()函数的代码清单

下面是 showPic()函数完整的代码清单：

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
}
```

接下来的任务是把这个 JavaScript 函数与我们的 HTML 文档结合起来。

4.3 JavaScript 函数的调用

我需要把刚编写出来 showPic()函数与图片库 HTML 文档结合起来。最简单的办法是把这个函数用一组<script>标签插入到那个文档的<head>部分，但我认为这种做法有点儿目光短浅：如果今后想把这同一函数用在多个页面上的话，我将不得不反复多次地进行剪贴操作。为今后考虑，更有远见的办法是先把这个函数存入一个外部文件，然后在每一份需要用到这个函数的 HTML 文档的<head>部分插入一个链接来引用这个外部文件。

以.js 作为文件扩展名，把这个函数存入一个文本文件。完全可以把这种文件命名为你们喜欢的任何东西，但我习惯于用这些文件所包含的函数的名字来命名它们——我给这个文件起的名字是 showPic.js。

就像我刚才决定把所有的图片集中存放在 images 子目录里那样，把所有的 JavaScript 脚本文件集中存放在一个子目录里也将是个好主意。我创建了一个名为 scripts 的子目录并把 showPic.js 文件保存到其中。

现在，需要在图片库 HTML 文档里插入一个链接来引用这个 JavaScript 脚本文件。我将把下面这条语句插入那份 HTML 文档的<head>部分（选择的插入位置是<head>标签之后）：

```
<script type="text/javascript" src="scripts/showPic.js"></script>
```

有了这条语句，把 showPic()函数与图片库 HTML 文档结合起来的任务就已经完成了一半——我还需要为 HTML 文档里的每个图片链接增加一个函数调用动作，否则 showPic()函数将永远也不会被调用。我将通过增加一个事件处理函数（event handler）的办法来完成这项工作。

事件处理函数

事件处理函数的作用是，在预定事件发生时让预先安排好的 JavaScript 代码开始执行，用术语来说就是 “触发一个动作”。例如，如果想在鼠标指针悬停在某个元素上时触发一个动作，就需要使用 onmouseover 事件处理函数；如果想在鼠标指针离开某个元素时触发一个动作，就需要使用 onmouseout 事件处理函数。在“JavaScript 美术馆”案例里，我想在用户点击某个链接时触发一个动作，所以需要使用 onclick 事件处理函数。

我想通过 onclick 事件处理函数去触发的动作是调用 showPic()函数，而要想调用这个函数，就必须向它传递一个参数：一个带有 href 属性的元素节点。在图片库 HTML 文档里，当我把 onclick 事件处理函数嵌入一些链接时，我需要把那些链接本身用作 showPic()函数的参数。

有个非常简单但又非常有效的办法可以做到这一点：使用 JavaScript 语言中的 this 关键字。这个关键字的含义是“这个对象”。具体到正在讨论的这个例子，我将使用 this 来表示“这个<a>元素节点”：

```
showPic(this)
```

综上所述，我将使用 onclick 事件处理函数去调用 showPic(this)方法。使用事件处理函数调用 JavaScript 代码的语法如下所示：

```
event = "JavaScript statement(s)"
```

请注意，在如上所示的语法里，JavaScript 代码是包含在一对引号之间的：我们可以把任意数量的 JavaScript 语句放在这对引号之间，只要把各条语句用分号隔开即可。

下面这条语句将完成“使用 onclick 事件处理函数调用 showPic(this)方法”的任务：

```
onclick = "showPic(this);"
```

不过，如果只把上面这个事件处理函数插入到 HTML 文档中的链接里去的话，将遇到这样一个问题：在 onclick 事件发生时，不仅 showPic(this)函数将被调用，链接被点击时的默认行为也将发生。这意味着用户还是会被带到另外一个图片查看窗口里去，而这是我不希望发生的事情。我需要阻止这种默认行为的发生。

要想达到这一目的，我们必须对事件处理函数的工作机制有进一步的了解：在给某个元素添加了事件处理函数后，一旦发生预定事件，相应的 JavaScript 代码就会得到执行；那些 JavaScript 代码可以返回一个结果，而这个结果将被传递回那个事件处理函数。例如，我们可以给某个链接添加一个 onclick 事件处理函数，并让这个处理函数所触发的 JavaScript 代码返回布尔值 true 或 false。这样一来，当这个链接被点击时，如果那段 JavaScript 代码返回给 onclick 事件处理函数的值是 true，onclick 事件处理函数将认为“这个链接被点击了”；反之，如果那段 JavaScript 代码返回给 onclick 事件处理函数的值是 false，onclick 事件处理函数将认为“这个链接没有被点击”。

可以通过下面这个简单测试去验证这一结论：

```
<a href="http://www.example.com" onclick="return false;">Click me</a>
```

当点击这个链接时，因为 onclick 事件处理函数所触发的 JavaScript 代码返回给它的值是 false，所以这个链接在被点击时的默认行为将不会发生。

同样道理，如果像下面这样在 onclick 事件处理函数所触发的 JavaScript 代码里增加一条 return false 语句的话，我就可以不让用户被他们所点击的链接带到另外一个图片查看窗口里去：

```
onclick = "showPic(this); return false;"
```

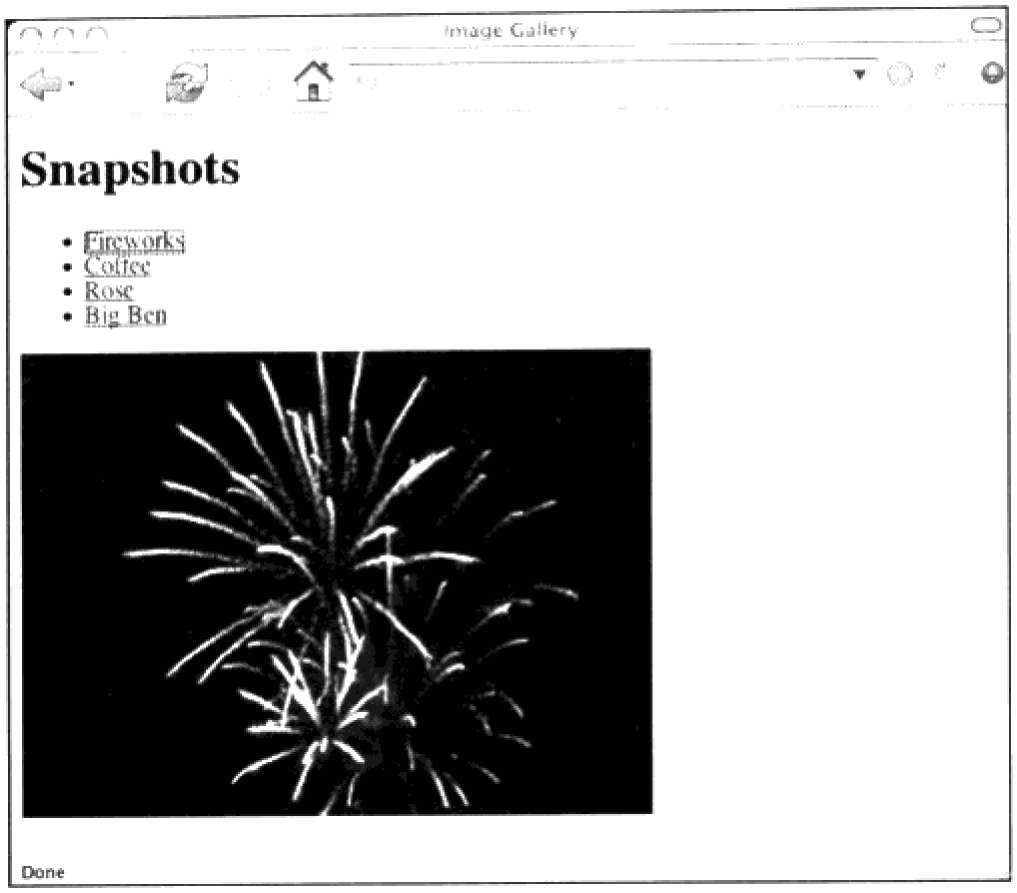
下面是最终完成的 onclick 事件处理函数在图片库 HTML 文档里的样子：

```
<li>
    <a href="images/fireworks.jpg" onclick="showPic(this);
➥ return false;" title="A fireworks display">Fireworks</a>
</li>
```

接下来，只要在图片库 HTML 文档里把这个 onclick 事件处理函数添加到每个链接上即可。这当然有些麻烦，但眼下只能这么做——我将在后面的内容里向大家介绍一个可以避免这种麻烦的办法。下面是图片库 HTML 文档在我以手动方式把这个 onclick 事件处理函数添加到各个有关链接上之后的样子：

```
<li>
    <a href="images/fireworks.jpg" onclick="showPic(this);
➥ return false;" title="A fireworks display">Fireworks</a>
  </li>
  <li>
    <a href="images/coffee.jpg" onclick="showPic(this);
➥ return false;" title="A cup of black coffee">Coffee</a>
  </li>
  <li>
    <a href="images/rose.jpg" onclick="showPic(this); return false;"
➥ title="A red, red rose">Rose</a>
  </li>
  <li>
    <a href="images/bigben.jpg" onclick="showPic(this); return false;"
➥ title="The famous clock">Big Ben</a>
  </li>
```

现在，如果把这个页面加载到 Web 浏览器里，你们将看到一个能够正常工作的“JavaScript 美术馆”：在如下所示的页面里，不管在图片列表里的哪个链接上点击鼠标，都将在这同一个页面里看到相应的图片。



4.4 对 JavaScript 函数进行功能扩展

在同一个网页上切换显示不同的图片并不是什么新鲜事。有着这类效果的网页和脚本早在W3C 推出它们的标准化 DOM 和 JavaScript 语言之前就已经出现了，如今更是得到了广泛的流行。

在这种情形下，如果你们想让自己与众不同，就必须另辟蹊径——你们有没有想过在同一个网页上切换显示不同的文本？我可不是在开玩笑，利用 JavaScript 语言和 DOM，确实可以做到这一点。

图片库 HTML 文档里的每个图片链接都有一个 `title` 属性。我想把这个属性的值提取出来并让它们伴随相应的图片一同显示在网页上。`title` 属性的值可以用 `getAttribute()`方法轻而易举地提取出来，如果把下面这条语句添加到 `showPic()`函数的开头的话，就可以把 `title` 属性的值保存到一个变量里：

```
var titletext = whichPic.getAttribute("title");
```

解决了提取 `title` 属性值的问题，还需要把它插入到 HTML 文档中的适当位置。为了完成这一工作，我需要用到几个新的 DOM 属性。

4.4.1 childNodes 属性

`childNodes` 属性让我们可以从给定文档的节点树里把任何一个元素的所有子元素检索出来。

childNodes 属性将返回一个数组，这个数组包含给定元素节点的全体子元素：

```
element.childNodes
```

假设我们需要把某个文档的 body 元素的全体子元素检索出来。首先，我们将使用 getElementsByTagName()方法来得到 body 元素。因为每份文档只有一个 body 元素，所以它将是 getElementsByTagName("body")方法所返回的数组中的第一个（也是唯一一个）元素：

```
var body_element = document.getElementsByTagName("body")[0];
```

现在，变量 body_element 已经指向了那个文档的 body 元素。接下来，可以用如下所示的语法记号把 body 元素的全体子元素检索出来：

```
body_element.childNodes
```

写出这个记号显然要比敲入下面这个长长的字符串要简明得多：

```
document.getElementsByTagName("body")[0].childNodes
```

顺便说一句，body 元素还有一个更简单的专用记号：

```
document.body
```

现在，已经知道如何把 body 元素的全体子元素检索出来了，我们来看看这些信息的用途。

首先，我们可以精确地查出 body 元素有多少个子元素。因为 childNodes 属性返回的是一个数组，所以可以用数组数据类型的 length 属性查出它所包含的元素的个数：

```
body_element.childNodes.length;
```

请把下面这个小函数添加到 showPic.js 文件里：

```
function countBodyChildren() {
  var body_element = document.getElementsByTagName("body")[0];
  alert (body_element.childNodes.length);
}
```

这个简单的小函数将弹出一个 alert 对话框，其显示内容是 body 元素的子元素的总数。

我想让这个函数在页面加载时执行，而这需要使用 onload 事件处理函数。把下面这条语句添加到代码段的末尾：

```
window.onload = countBodyChildren;
```

这条语句将在页面加载时调用 countBodyChildren 函数。

在 Web 浏览器里刷新 gallery.html 文件。你们将看到一个 alert 对话框，其显示内容是 body 元素的子元素的总数。那个数字很可能会让你们大吃一惊。

4.4.2 nodeType 属性

根据 gallery.html 文件的结构，body 元素应该只有 3 个子元素：一个 h1 元素、一个 ul 元

素和一个 imag 元素。可是，countBodyChildren()函数给出来的数字却远大于此。之所以会这样是因为文档树的节点类型并非只有元素节点一种。

由 childNodes 属性返回的数组包含着所有类型的节点，除了所有的元素节点，所有的属性节点和文本节点也包含在其中。事实上，文档里几乎每一样东西都是一个节点——甚至连空格和换行符都会被解释为节点，而它们也全都包含在 childNodes 属性所返回的数组当中。

因此，countBodyChildren()函数的返回结果才会是一个相当大的数字。

还好，我们可以利用 nodeType 属性来区分文档里的各个节点。这个属性可以让我们知道自己正在与哪一种节点打交道。不过，这个属性返回的是一个数字而不是像“element”或“attribute”那样的英文字符串。

下面是 nodeType 属性的调用语法：

node.nodeType

为了验证这一点，请把 countBodyChildren()函数中的 alter 语句替换为下面这条语句，这样一来，我们就可以知道 body_element 元素的 nodeType 属性了：

```
alert(body_element.nodeType);
```

在 Web 浏览器里刷新 gallery.html 文件，将看到一个显示数字“1”的 alert 对话框。换句话说，元素节点的 nodeType 属性值是 1。

nodeType 属性总共有 12 种可取值，但其中仅有 3 种具有实用价值：元素节点、属性节点和文本节点的 nodeType 属性值分别是 1、2 和 3。

- **元素节点**的 nodeType 属性值是 1。
- **属性节点**的 nodeType 属性值是 2。
- **文本节点**的 nodeType 属性值是 3。

这个意味着，可以让我们的函数只对某种特定类型的节点进行处理。例如，完全可以编写出一个只对元素节点进行处理的函数。

4.4.3 在 HTML 文档里增加一段描述性文本

为了让我的“JavaScript 美术馆”变得与众不同，我决定给它增加一个文本节点。我想在显示图片时把这个文本节点的值替换为一个来自某个属性节点(某个图片链接的 title 属性)的值。

首先，需要为打算显示的文本安排显示位置。我将在 gallery.html 文件里增加一个新的文本段。我决定把它安排在<img>标签之前。我将为它设置一个独一无二的 id 值，这样就能在 JavaScript 函数里方便地引用它了：

```
<p id="description">Choose an image.</p>
```

上面这条语句将把<p>元素的 id 属性设置为 description（描述），这个单词可以让这个元素的用途一目了然。如下所示，包含在此元素里的文本现在是“Choose an image.”，但我打算在切换显示图片时把这段文本同时替换为相应的描述性文字。



我想达到的预期效果是：在“JavaScript 美术馆”里的某个图片链接被点击时，不仅要把“占位符”图片替换为那个链接的 href 属性所指向的图片，还要把这段文本同时替换为那个链接的 title 属性值。为了实现这一想法，需要对 showPic()函数做一些改进。

4.4.4　用 JavaScript 代码改变<p>元素的文本内容

为了把“JavaScript 美术馆”里的图片说明在某个图片链接被点击时，动态地替换为那个链接的 title 属性值，我需要对 showPic()函数做一些修改。

下面是 showPic()函数现在的样子：

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
}
```

首先，我需要在 showPic()函数里增加一些语句来提取 whichpic 对象的 title 属性值。我将把提取到的 title 属性值存入 text 变量。这些事可以轻而易举地利用 getAttribute()方法完成：

```
var text = whichpic.getAttribute("title");
```

接下来，为了让自己能在代码里方便地引用那段 id 属性值等于 description 的文本，我决定创建一个新的变量来存放它：

```
var description = document.getElementById("description");
```

下面是 showPic()函数在我给它增加了上述两个变量之后的样子：

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
  var text = whichpic.getAttribute("title");
  var description = document.getElementById("description");
}
```

我们的下一个任务是实现文本的切换显示效果。

4.4.5 nodeValue 属性

如果想改变某个文本节点的值，那就使用 DOM 提供的 nodeValue 属性，它的用途正是检索（和设置）节点的值：

node.nodeValue

但这里有个大家必须注意的细节：在用 nodeValue 属性检索 description 对象的值时，你得到的并不是包含在这个段落里的文本。可以用下面这条 alert 语句来验证这一点：

```
alert (description.nodeValue);
```

这个调用将返回一个 null 值。<p>元素的 nodeValue 属性值是一个空值，而我们这里需要的是<p>元素所包含的文本的值。

包含在<p>元素里的文本是另一种节点，它在 DOM 里是<p>元素的第一个子节点。换句话说，如果想获得<p>元素的文本内容，就必须检索它的第一个子节点的 nodeValue 属性值。

下面这条 alert 语句可以找到我们想要的内容：

```
alert(description.childNodes[0].nodeValue);
```

这个调用的返回值才是我们正在寻找的“Choose an image.”。这个值来自 childNodes[]数组的第一个（下标是 0）元素。

4.4.6 firstChild 和 lastChild 属性

数组元素 childNodes[0]有个更直观易读的同义词。无论何时何地，只要需要访问 childNodes[]数组的第一个元素，我们都可以把它写成 firstChild，这个记号本身是一个属性：

node.firstChild

这种用法与下面这个语法记号完全等价：

```
node.childNodes[0]
```

单词“firstChild”不仅更加简短，还更加具有可读性。

DOM 还提供了一个与之对应的 lastChild 属性：

```
node.lastChild
```

这个语法记号代表着 childNodes[]数组的最后一个元素。如果不想通过 lastChild 属性去访问这个节点，我们将不得不使用如下所示的语法记号：

```
node.childNodes[node.childNodes.length-1]
```

与简明易懂的 lastChild 相比，这么复杂的语法记号恐怕没人会喜欢。

4.4.7　利用 nodeValue 属性刷新<p>元素的文本内容

现在，我们将回到 showPic()函数。我将对 id 属性值等于 description 的<p>元素所包含的文本节点的 nodeValue 属性进行刷新。

具体到这个 id 属性值等于 description 的<p>元素，因为它只有一个子节点，所以选用 firstChild 属性或是选用 lastChild 属性的效果是完全一样的。既然如此，我决定选用 firstChild 属性。

我可以把 4.4.5 节里的 alter 语句改写为如下所示的样子：

```
alert(description.firstChild.nodeValue);
```

两条语句的效果完全一样（都将显示“Choose an image.”消息），但这里的代码显然更容易阅读和理解。

nodeValue 属性的用途并非仅限于此。我们不仅可以用它来检索某个节点的值，还可以用它来设置某个节点的值；后一种用法正是我目前最需要的。

还记得刚才在 showPic()函数里的 text 变量吗？当“JavaScript 美术馆”页面上的某个图片链接被点击时，showPic()函数会把这个链接的 title 属性值传递给 text 变量。而我现在将用 text 变量去刷新 id 值等于 description 的那个<p>元素的第一个子节点的 nodeValue 属性值，如下所示：

```
description.firstChild.nodeValue = text;
```

下面是我为了改进 showPic()函数而给它添加的三条新语句：

```
var text = whichpic.getAttribute("title");
var description = document.getElementById("description");
description.firstChild.nodeValue = text;
```

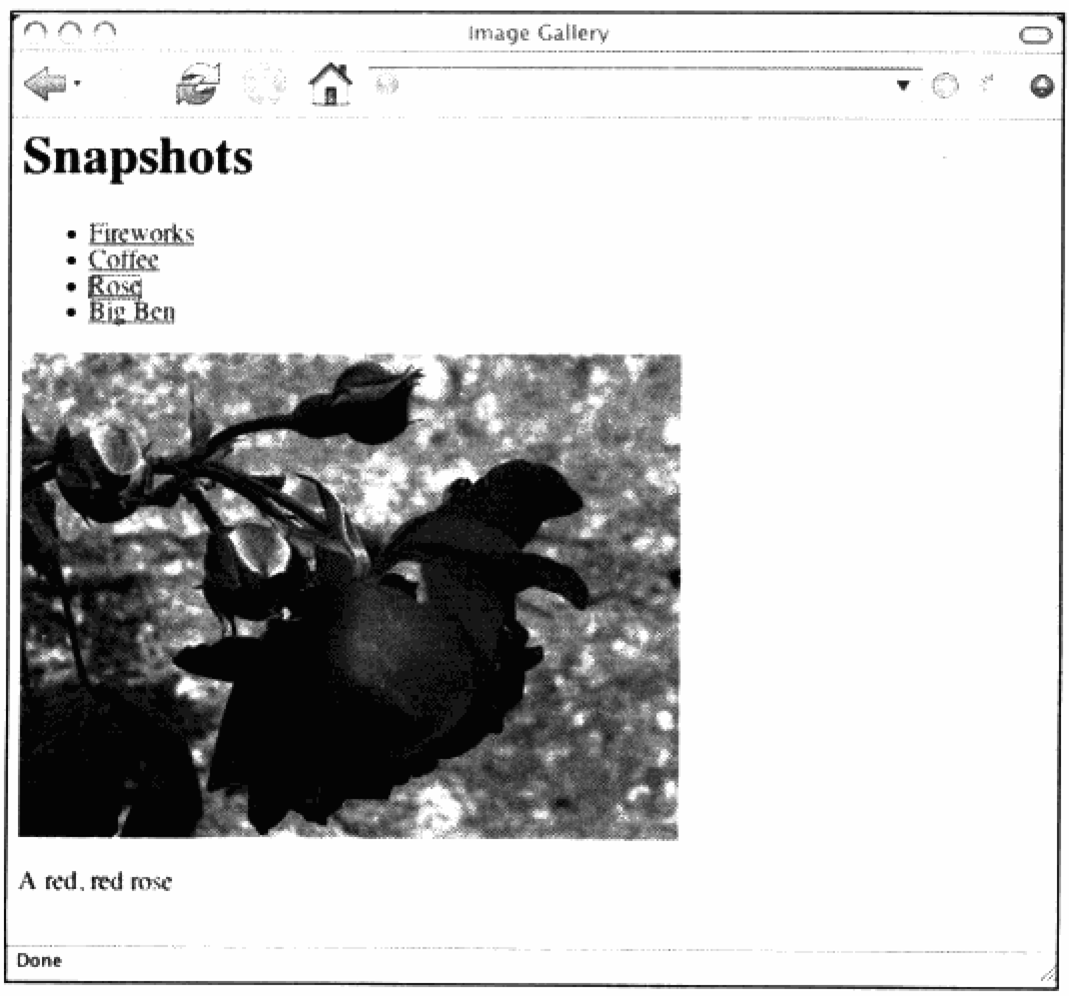
如果用日常生活中的语言来说，这三条语句的含义依次是：

- 当“JavaScript 美术馆”页面上的某个图片链接被点击时，这个链接的 title 属性值将被提取并保存到 text 变量中。
- 找到那个 id="description"的<p>元素，并把这个对象保存到变量 description 里去。
- 把 description 对象的第一个子节点的 nodeValue 属性值设置为变量 text 的值。

下面是完成上述改进后的 showPic()函数的代码清单：

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
  var text = whichpic.getAttribute("title");
  var description = document.getElementById("description");
  description.firstChild.nodeValue = text;
}
```

如果想测试一下这些扩展功能，请把改进后的 showPic()函数存入 showPic.js 文件，然后在浏览器里刷新 gallery.html 文档。现在，点击这个网页上的某个图片链接时，你们将看到两种效果：“占位符”图片将被替换为这个链接所指向的一张新图片，同时图片下方的描述性文字也将被替换为这个链接的 title 属性值，如下所示。



你们可以在 http://friendsofed.com/网站上找到“JavaScript 美术馆”脚本文件和 HTML 文档。我在示例中用到的图片也可以在那里找到，但我建议大家找一些自己的图片来测试这个脚本，那样会更有意思。

如果想让这个“JavaScript 美术馆”变得更美观，可以再给它增加一个像下面这样的样式表：

```
body {
  font-family: "Helvetica","Arial",sans-serif;
  color: #333;
  background-color: #ccc;
  margin: 1em 10%;
}
h1 {
  color: #333;
  background-color: transparent;
}
a {
  color: #c60;
  background-color: transparent;
  font-weight: bold;
  text-decoration: none;
}
ul {
  padding: 0;
}
li {
  float: left;
  padding: 1em;
  list-style: none;
}
```

请把这些 CSS 代码存入 layout.css 文件，并把这个文件存放到 styles 子目录里。然后，就可以在 gallery.html 文档的<head>部分用一个<link>标签来引用这个文件了，如下所示：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Image Gallery</title>
  <script type="text/javascript" src="scripts/showPic.js"></script>
  <link rel="stylesheet" href="styles/layout.css"
➥ type="text/css" media="screen" />
</head>
<body>
  <h1>Snapshots</h1>
  <ul>
    <li>
      <a href="images/fireworks.jpg" title="A fireworks display"
```

```
➥ onclick="showPic(this); return false;">Fireworks</a>
    </li>
    <li>
      <a href="images/coffee.jpg" title="A cup of black coffee"
➥ onclick="showPic(this); return false;">Coffee</a>
    </li>
    <li>
      <a href="images/rose.jpg" title="A red, red rose"
➥ onclick="showPic(this); return false;">Rose</a>
    </li>
    <li>
      <a href="images/bigben.jpg" title="The famous clock"
➥ onclick="showPic(this); return false;">Big Ben</a>
    </li>
  </ul>
  <img id="placeholder" src="images/placeholder.gif"
➥ alt="my image gallery" />
  <p id="description">Choose an image.</p>
</body>
</html>
```

下面是“JavaScript 美术馆”在经过上面那个简单的样式表修饰之后的显示效果。



4.5 小结

本章向大家介绍了一个简单的 JavaScript 应用案例。在实现这个案例的过程中，我们还向大家介绍了 DOM 提供的几个新属性，它们是：

- childNodes
- nodeType
- nodeValue
- firstChild
- lastChild

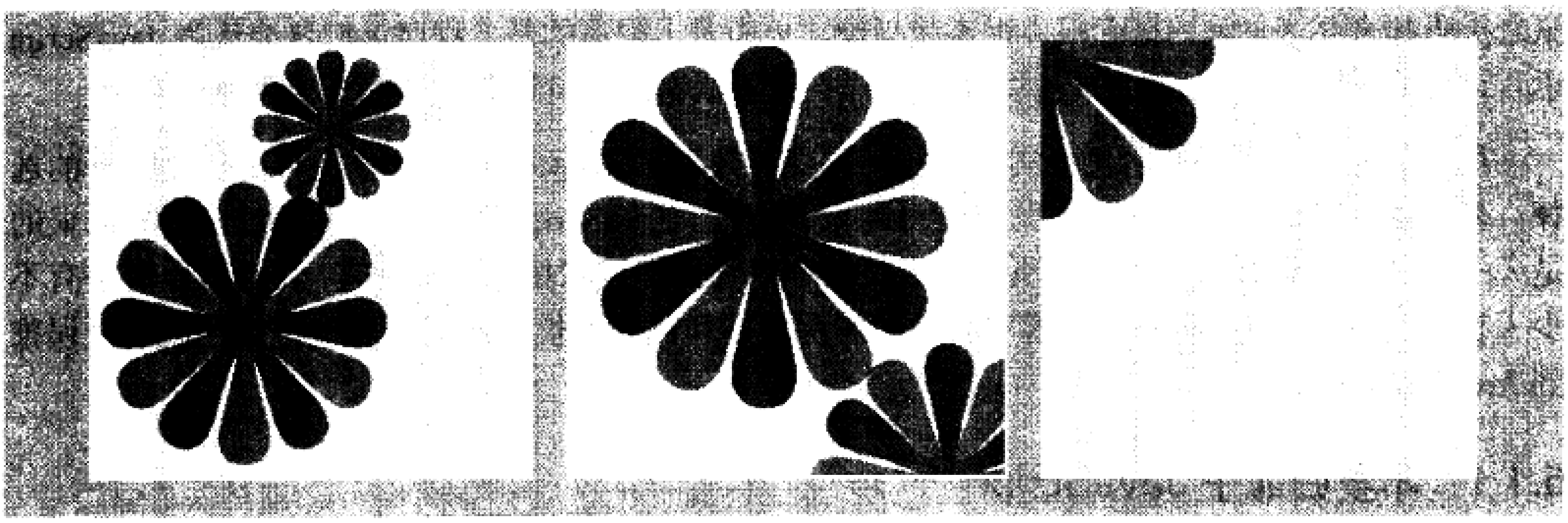
本章的学习重点有两个：一是如何利用 DOM 所提供的方法去编写图片库脚本；二是如何利用事件处理函数把 JavaScript 代码与网页集成在一起。

从表面上看，我们的“JavaScript 美术馆”已经大获成功，但它实际上还有许多地方值得改进，而那将是随后两章里的讨论重点。

在下一章中，我将向大家介绍一些 JavaScript 脚本编程方面的原则和良好习惯，我希望它们能让你们领悟这样一个道理：通往终点的过程与终点本身同样重要。

在那之后，我将向你们演示如何把那些编程原则和良好习惯应用到“JavaScript 美术馆”案例上。

第5章 JavaScript编程原则和良好习惯



本章内容

- 预留退路：确保网页在没有 JavaScript 的情况下也能正常工作。
- 分离 JavaScript：把网页的结构和内容与 JavaScript 脚本的动作行为分开。
- 向后兼容性：确保老版本的浏览器不会因为你的 JavaScript 脚本而死机。

JavaScript 语言和 DOM 构成了一个功能非常强大的组合，但问题的关键是你能否恰到好处地运用它们所提供的功能。本章将向大家介绍一些 JavaScript 编程原则和良好习惯，它们可以帮助你们保证编写出来的脚本不会与你们的愿望背道而驰。

5.1 不要误解 JavaScript

JavaScript 的名声并不好。从它诞生时起，就有许多程序员老手对它持怀疑态度，而 Netscape 公司把其命名为"JavaScript"不仅没能帮助它打开局面，反而给它带来了许多负面影响：因为这个名字不可避免地会让人们把 JavaScript 与 Java 语言进行比较，并得出"名不副实"的结论。

有不少程序员本以为 JavaScript 能够像 Java 语言那样功能强大，但在试用 JavaScript 语言之后却大失所望。虽然它们在语法上颇为相似，但 JavaScript 实际上只是一种相对简单得多的语言。Java 语言为"面向对象"的程序设计提供了良好的支持，JavaScript 语言却停留在"面向过程"的层次上。

公平地讲，JavaScript 语言并不是不能用于编写“面向对象”的程序，但可惜的是这个优点被“JavaScript”这个名字给抹杀了——在将其与 Java 对比之后，有不少程序员得出了“JavaScript 只不过是另一种脚本语言而已”的结论。

JavaScript 语言在网页设计人员那里也受到了冷遇，但原因恰好相反：就在程序员们认为这种语言过于简单的同时，网页设计人员却因为它的名字里有“Java”而想当然地认为它难以学习和掌握。

易学易用的 HTML 语言，让许多网页设计人员对 JavaScript 语言中诸如对象、函数、数组、变量之类听起来高深莫测的概念根本提不起兴趣。更为雪上加霜的是，程序员们普遍认为 JavaScript 语言过于简单这一事实，让一些原本有兴趣学习这种语言的网页设计人员也变得犹豫起来。

尽管如此，当需要在网页里实现一些动态交互效果时，JavaScript 至少在目前仍是唯一的选择。基于这种考虑，笔者认为它只会变得越来越流行。JavaScript 语言本身并无过错，犯错误的是人——之所以会有那么多的网页设计人员和程序员们轻视和抵触这种语言，其根本原因是有不少人在编写 JavaScript 脚本时未遵守有关的编程原则和良好习惯，我们不应该把自己犯的错误推到 JavaScript 身上。

5.1.1 不要归罪于 JavaScript

易学易用的技术就像一把双刃剑。因为容易学习和掌握，它们往往会在很短的时间内就为人们广泛接受。但“易学易用”往往意味着缺乏高水平的质量控制措施。

HTML 语言就是一个很好的例子。万维网（World Wide Web）之所以会出现爆炸性的增长，HTML 语言易学易用的特点是无可否认的原因之一。人们只须花费很短时间就能掌握 HTML 语言的基本知识，并迅速地创建出各种各样的网页。事实上，随着“所见即所得”网页设计工具的出现和流行，有些人可能连一条 HTML 语句都没有见过就成为了网页设计大军中的一员。

因此而产生的不良后果之一是，绝大多数网页都编写得很糟糕，甚至不做数据合法性检查。因此，软件厂商不得不让它们的浏览器软件以尽可能宽松的方式去处理网页。每种浏览器软件都有相当一部分代码专门用来处理那些含糊不清的 HTML 文档，以及猜测网页的创作者们到底想如何呈现网页。

理论上讲，在如今的 Web 上有数以十亿计的 HTML 文档；但在实践中，这些文档中只有少部分有着完全符合有关规范的结构。这种历史遗留问题使得 XHTML 和 CSS 等新技术在 Web 上的推广和应用遇到了很大的阻力。

易学易用的 HTML 语言既是万维网的福音，又是它的噩梦。

与 HTML 语言相比，JavaScript 语言的境遇要凄凉得多。如果 JavaScript 代码不符合语法规定，JavaScript 解释器（对 Web 应用而言就是浏览器）将拒绝执行它们并报错；而浏览器在遇到不符合语法规定的 HTML 代码时，则会千方百计地将其呈现出来。

令人遗憾的是，在如今的 Web 上有一大批质量低劣的 JavaScript 代码。

许多网页设计者并不花费时间去学习 JavaScript 语言，而是把一些现成的 JavaScript 代码直接剪贴到他们的 HTML 文档里以使网页变得更加丰富多彩。事实上，JavaScript 语言诞生后不久，市场上就出现了许多能让人们把 JavaScript 代码片段嵌入或关联到 HTML 文档的“所见即所得”的网页设计工具。

其实，即使没有那些“所见即所得”的网页设计工具，把 JavaScript 代码嵌入或关联到 HTML 文档也不是难事。有许多网站和书刊专门提供各种现成的 JavaScript 函数并号称自己提供的 JavaScript 函数便于使用。一时间，“剪切和粘贴”成了编写 JavaScript 脚本的代名词。

不幸的是，许多现成的 JavaScript 函数对问题考虑得并不周全：从表面上看，它们都能完成自己的任务并给网页带来新颖动人的交互效果；但在实际应用中，它们当中只有很少一部分能够在 JavaScript 被禁用时对网页的行为做出妥善的安排。很多时候，一旦浏览器不支持或禁用了 JavaScript 解释功能，那些质量低劣的脚本就会导致用户无法浏览相应的网页甚至整个网站。因为这类问题（用术语来说，就是“某某网站/网页的可访问性很差”）频繁发生，没过多久，“JavaScript”就在许多人的脑海里成为了“网页无法访问”的同义词。

事实上，JavaScript 与“网页无法访问”无任何必然的联系，网页能否访问完全取决于 JavaScript 脚本的编写质量，也就是取决于脚本的编写者。这是一个“工欲善其事，必先利其器”的典型例子。

5.1.2 Flash 的遭遇

客观地讲，没有不好的技术，只有没有用好的技术。JavaScript 的坎坷遭遇让我不禁想起了另一种被人们滥用的技术：由 Micromedia 公司研发的 Flash。

现在，有不少人一提起 Flash 就会想起烦人的前导页面、超长的下载时间和随时都有可能出问题的浏览体验。这些恶劣印象其实与 Flash 毫不相干，它们都是由那些质量低劣的 Flash 视频脚本造成的。

把 Flash 与超长的下载时间联系在一起很不公平，因为制作短小精悍的矢量图形和视频片段本是 Flash 技术的强项之一。利用 Flash 技术制作一些视频片段来介绍自己的网站是一个很好的创意，但当这种做法成为一种潮流时，这类视频片段的数量越来越多、体积也越来越大，网页的下载时间也不可避免地变得越来越长。此时，Flash 要想洗刷掉自己身上的恶名谈何容易。

类似地，JavaScript 本是一种能让网页变得易于访问的技术，而使其具有“JavaScript 会让网站变得难以制作和访问”的恶劣印象的是人们对这种技术的滥用。

正如物理学中的运动与惯性定律所描述的那样，如果人们在开始使用一种新技术时没有经过深思熟虑，而这种新技术又很快地成为了一种潮流，则纠正在早期阶段养成的坏习惯将会非常困难。

我敢说，之所以会有那么多的网站迫不及待地在网页上嵌入一些毫无必要的 Flash 视频片段，

是因为“大家都有，所以我也要有”的心理而不是因为实际应用的需要。既然别人的网页上有 Flash 动画，那么我的网页上也要有 Flash 动画——有无必要的问题已无人问津了。

JavaScript 也遭遇到了类似的命运：人们只关心自己的网页里有没有 JavaScript 代码，根本不去考虑那些现成的（尤其是那些由“所见即所得”网页设计工具生成的）JavaScript 函数本身有没有漏洞，以及它们会不会给网页带来负面影响。JavaScript 代码被人们剪贴来、剪贴去，结果弄得网上到处都是似是而非的 JavaScript 网页。在潮流的影响下，几乎没人想到应该首先检查一下那些现成的 JavaScript 函数是否还需要改进。

5.1.3 质疑

不管你们想通过 JavaScript 去改变哪个网页的行为，都必须三思而后行。首先要确认：为这个网页增加这种额外的行为是否确有必要？

网站对 JavaScript 的滥用已经持续了相当长的时间，因为滥用 JavaScript 而给自己带来种种麻烦的网站也绝不是少数。例如，可以用 JavaScript 脚本让浏览器窗口在屏幕上四处移动，甚至让浏览器窗口产生振动效果。

在所有的 JavaScript 特效当中，最臭名昭著的莫过于那些在人们打开网页时弹出的广告窗口，它们其实是 JavaScript 脚本在当前浏览器窗口下生成的子窗口。这些弹出广告进一步加重了 JavaScript 的坏名声。有不少用户为此干脆彻底禁用了 JavaScript。浏览器厂商也在各自的产品里提供了种种内建的广告过滤机制来解决这一问题。

弹出窗口是一个典型的滥用 JavaScript 的例子。从技术上讲，弹出窗口本身是一项很实用的功能，它解决了网页设计工作中的一个难题：如何向用户发送信息。但在实践中，频繁弹出的广告窗口却让用户不胜其烦。那些弹出窗口必须由用户去关闭，而这往往会形成一种拉锯战——用户刚关闭了一个广告窗口，屏幕上又弹出一个。

那么，这一功能要如何使用户受益呢？

令人感到欣慰的是，这一问题正越来越受到人们的关注，遵循“用户至上”的原则设计网页的网站也越来越多。笔者认为，从长远看，对付弹出广告的其他办法都不如简单地遵循这一原则来得持久和有效。

如果要使用 JavaScript，就要确认：这么做会对用户的浏览体验产生怎样的影响？还有个更重要的问题：如果用户的浏览器不支持 JavaScript 该怎么办？

5.2 预留退路

记住，网站的访问者完全有可能使用的是不支持 JavaScript 的浏览器，还有一种可能是虽然浏览器支持 JavaScript，但用户已经禁用它了（比如，因为讨厌看到弹出广告）。如果没有考虑到这种情况，人们在访问你们的网站时就有可能遇到各种各样的麻烦，并因此不再来访问你们的网站。

如果正确地使用了 JavaScript 脚本，就可以让访问者在他们的浏览器不支持 JavaScript 的情况下仍能顺利地浏览你们的网站。这就是所谓的**预留退路**（graceful degradation），就是说，虽然某些功能无法使用，但最基本的操作仍能顺利完成。

我们来看一个在新窗口里打开一个链接的例子。别担心：我们将要讨论的并不是在网页加载时弹出新窗口。我将讨论的是在用户点击某个链接时弹出一个新窗口。

这其实是一项相当实用的功能。例如，在许多电子商务网站的结算页面上都有一些指向服务条款或是邮寄费用表的链接，与其让用户在点击这些链接时被带离当前页面，让用户仍停留在当前页面，并用一个弹出窗口来显示相关信息无疑是一种更好的解决方案。

注意 应该只在有绝对必要的情况下才使用弹出窗口，因为这将牵涉到网页的可访问性问题：浏览器不支持弹出窗口，浏览器的弹出窗口支持功能已被用户禁用，用户使用的屏幕读取软件无法向用户说明弹出了窗口，等等。因此，如果网页上的某个链接将弹出新窗口，最好在这个链接本身的文字中予以说明。

JavaScript 使用 `window` 对象的 `open()`方法来创建新的浏览器窗口。这个方法有三个参数：

window.open(url,name,features)

这三个参数都是可选的。第一个参数是想在新窗口里打开的那份文档的 URL 地址。如果省略这个参数，屏幕上将弹出一个空白的浏览器窗口。

第二个参数是新窗口的名字。可以在代码里通过这个名字与新窗口进行通信。

最后一个参数是一个以逗号分隔的字符串，其内容是新窗口的各种属性。这些属性包括新窗口的尺寸（宽度和高度）以及新窗口被激活或禁用的各种浏览功能（工具条、菜单条、初始显示位置，等等）。对于这个参数应该掌握以下原则：新窗口的浏览功能要少而精。

`open()`方法是 BOM 的一个典型应用案例：它的功能既不依赖于文档的内容，对文档的内容也无任何影响。这个方法只与此时此刻的用户浏览环境（具体到要讨论的例子，就是当前 `window` 对象）有关。

下面这个函数是 `window.open()`方法的一种典型应用：

```
function popUp(winURL) {
  window.open(winURL,"popup","width=320,height=480");
}
```

这个函数将打开一个 320 像素宽、480 像素高的新窗口“popup”。因为我在这个函数里已为新窗口命名，所以当把新的 URL 地址传递给此函数时，这个函数将把新窗口里的现有文档替换为新 URL 地址处的文档，而不是再去创建一个新窗口。

我将把这个函数存入一个外部文件。因此，当需要在某个网页里使用此函数时，只要在这个

网页的<head>部分用一个<script>标签导入那个外部文件即可。函数本身不会对网页的可访问性产生任何影响——会影响到网页的只是：我将如何使用此函数。

5.2.1 "javascript:" 伪协议

popUp()函数还可以通过伪协议（pseudo-protocol）来调用。

"真"协议特指那些用来在因特网上的两台计算机之间传输各种数据包的标准化通信机制，如 http://、ftp://等，伪协议则是人们对非标准化通信机制的统称。"javascript:"伪协议让我们可以通过一个链接来调用 JavaScript 函数。

下面是通过"javascript:"伪协议调用 popUp()函数的具体做法：

```
<a href="javascript:popUp('http://www.example.com/');">Example</a>
```

这条语句在支持"javascript:"伪协议并启用了 JavaScript 功能的浏览器中运行正常；不支持这种伪协议的浏览器则会去尝试打开那个链接但失败；支持这种伪协议但禁用了 JavaScript 功能的浏览器会什么也不做。

总之，在 HTML 文档里通过"javascript:"伪协议调用 JavaScript 代码的做法非常不好。

5.2.2 内嵌的事件处理函数

我们已经在"JavaScript 美术馆"里见识过事件处理函数的用途和用法了：把 onclick 事件处理函数作为属性嵌入<a>标签，该处理函数将在 onclick 事件发生时调用图片切换函数。

这个技巧同样可以用来调用 popUp()函数。但当在某个链接里用 onclick 事件处理函数去打开新窗口时，这个链接的 href 属性似乎没有什么用处——与这个链接有关的重要信息已经都包括在它的 onclick 属性里了。这也正是我们经常会看到如下所示的链接的原因：

```
<a href="#" onclick="popUp('http://www.example.com/');
➥ return false;">Example<a>
```

因为在上面这条 HTML 指令里使用了 return false 语句，这个链接不会真的被打开。"#"符号是一个仅供文档内部使用的链接记号（单就这条指令而言，"#"是未指向任何目标的内部链接）。在某些浏览器里，"#"链接指向当前文档的开头。把 href 属性的值设置为"#"只是为了创建一个空链接。实际工作将全部由 onclick 属性负责完成。

很遗憾，这个技巧与用"javascript:"伪协议调用 JavaScript 代码的做法同样糟糕，因为它们都没有预留退路。如果用户已经禁用了浏览器的 JavaScript 功能，这样的链接将毫无用处。

5.2.3 有何不好

或许你对我反复强调"预留退路"有些不解：让那些不支持或禁用了 JavaScript 功能的浏览器也能顺利地访问你的网站真的那么重要吗？

请想像一下，有个访问者来到了你的网站，他总是在浏览 Web 时同时禁用图像和 JavaScript。你们肯定会认为这样的用户如今已非常少见，而事实也正是如此。但这个访问者非常重要。

你们刚才想像的那个用户是一个搜索机（searchbot）。搜索机是一种自动化的程序，它们浏览 Web 的目的是为了把各种网页添加到搜索引擎的数据库里。各大搜索引擎都有与此类似的程序。目前，只有极少数搜索机能够理解 JavaScript 代码。

如果你的 JavaScript 网页没有预留退路，它们在搜索引擎上的排名肯定会大受损害。

具体到 popUp()函数，为其中的 JavaScript 代码预留出退路是很简单的：在有关的链接里把 href 属性设置为真实存在的 URL 地址，让它成为一个有效的链接，如下所示：

```
<a href="http://www.example.com/"
➥ onclick="popUp('http://www.example.com'); return false;">Example</a>
```

因为 URL 地址出现了两次，上面这些代码显得有点冗长，但我们可以利用 JavaScript 语言提供的一个快捷方式把它改写得简明一些。JavaScript 语言中的 this 可以用来代表任何一种当前元素，所以可以用 this 和 getAttribute()方法提取出 href 属性的值，如下所示：

```
<a href="http://www.example.com/"
➥ onclick="popUp(this.getAttribute('href')); return
false;">Example</a>
```

老实说，上面这条语句没有精简多少。当前链接的 href 属性还有一个更简明的引用办法——使用由 DOM 提供的 this.href 属性：

```
<a href="http://www.example.com/"
➥ onclick="popUp(this.href); return false;">Example</a>
```

不管采用的哪种方法，重要的是 href 属性现在已经有了合法有效的值。与 href = "javascript:..."或 href = "#"相比，这几种变体的效果要好得多。

所以，在把 href 属性设置为真实存在的 URL 地址后，即使 JavaScript 已被禁用（或遇到了搜索机），这个链接也是可用的：虽然这个链接在功能上打了点儿折扣（因为它没有打开一个新窗口），但它并没有彻底失效。这是一个经典的“预留退路”例子。

在本书此前介绍的所有技巧当中，我认为这个技巧是最有用的，但它还有改进的余地。这个技巧最明显的不足是：每当需要打开新窗口时，就不得不把一些 JavaScript 代码嵌入 HTML 文档中。如果能把包括事件处理函数在内的所有 JavaScript 代码全都放在外部文件里，这个技巧将更加完善。

5.3 向 CSS 学习

此前，我曾以 JavaScript 和 Flash 为例，对技术会因为在诞生初期被人们滥用而造成恶劣后果的问题进行了讨论。我们可以从过去的失误里学到很多东西。

不过，还有一些技术是从一开始就被人们小心谨慎地使用着的。我们可以从它们那里学到更多的东西。

CSS（层叠样式表）是一项了不起的技术。CSS 可以让人们对网站设计工作中的各个方面做出严格细致的控制。从表面上看，CSS 技术并无新内容，CSS 可以做到的用<table>和<font>等标签也可以做到。CSS 技术的最大优点是，它把 Web 文档的内容结构（HTML 文档本身）和版面设计分成了互不影响的两大部分。

我们经常会遇到一些几乎每个元素都带有 style 属性的 Web 文档，而这是 CSS 技术最缺乏效率的用法之一。而真正能从 CSS 技术身上获得收益的方法，应该是，把样式全部转移到相关的外部文件里去。

与 JavaScript 和 Flash 相比，CSS 的“出生”日期要晚得多。或许是已经从滥用 JavaScript 和 Flash 的后果中吸取了教训的缘故，网页设计人员一开始使用 CSS 时就采用了一种深思熟虑、循序渐进的态度。

把文档的结构和样式划分为两大部分的 CSS 技术给每个人都带来了方便。如果你的工作是编写文档的内容，现在只要集中精力把文档的内容正确地标记出来就行了，用不着再与充斥着<table>和<font>等标签的模板打交道，也就用不着再担心会把文档的版面设计弄得一团糟。如果你的工作是设计网页的版面，现在只要集中精力把诸如颜色、字体和位置等在一些外部文件里设置妥当就行了，而无需再接触文档，你最多需添加类或是 id 属性。

作为 CSS 技术的突出优点，文档结构与文档样式的分离可以确保各有关网页都有预留退路。具备 CSS 支持的浏览器固然可以把网页呈现得美仑美奂，不支持或禁用了 CSS 功能的浏览器同样可以把网页的内容按照正确的结构显示出来。虽然各大搜索引擎的搜索机（如 Googlebot 等）不能理解 CSS 代码，但这丝毫不会妨碍它们浏览那些使用了 CSS 技术的网站。

在学习和使用 JavaScript 时，我们可以从 CSS 身上借鉴到很多东西。

循序渐进

在网页设计人员当中流传着这样一句格言：“内容就是一切”。如果没有内容，创建网站还有何用？

话虽如此，也不能简单地把原始内容发布到网上，而不加任何描述。内容需要用 HTML 或 XHTML 之类的标记语言来描述。在创建网站的时候，给内容加上正确的 HTML 标记是第一个步骤，或许也是最重要的步骤。我们可以修正那句格言为“标记良好的内容就是一切”。

只有正确地使用标记语言才能对内容做出准确的描述。标记语言中的各种标记负责提供诸如“这是列表项”、“这是文本段落”之类的信息。如果不使用<li>、<p>之类的标签，我们就很难把它们区分开来。

在给内容加上各种必要的标记后，我们还可以借助于各种 CSS 指令对内容的显示效果进行控制。CSS 指令构成了一个表示层。这个表示层就像是一张透明的彩色薄膜，可以包裹到文档的结构上，使文档的内容呈现出各种色彩。但即使去掉这个表示层，文档的内容也依然可以访问（只是缺乏色彩而已）。

所谓“循序渐进”就是用一些额外的信息层去包裹原始数据的做法。按照“循序渐进”原则创建出来的网页几乎，如果不是“全部”的话，都符合“预留退路”原则。

类似于 CSS，JavaScript 和 DOM 提供的所有功能也应该构成一个额外的、不影响文档结构和内容的指令层。CSS 代码负责提供关于“表示”的信息，JavaScript 代码负责提供关于“行为”的信息。行为层的应用方式与表示层一样。

要想获得最佳的“表示”效果，就应该把 CSS 代码从 HTML 文档里分离出来放在一些外部文件里。像下面这样把 CSS 代码混杂在 HTML 文档里也不是不可以，但这种做法弊大于利：

```
<p style="font-weight: bold; color: red;">
Be careful!
</p>
```

更值得推荐的办法是，先把样式信息存入一个外部文件，再在文档的<head>部分用<link>标签来调用这个文件：

```
.warning {
  font-weight: bold;
  color: red;
}
```

class 属性是样式与文档内容之间的联结纽带：

```
<p class="warning">
Be careful!
</p>
```

这显然更容易阅读和理解，而且样式信息也更容易修改了。例如，假设你在 100 个文档里使用了 warning 类来排版各种警告信息，而现在想统一改变那些警告信息的显示效果——比如把它们的颜色都从红色改为蓝色。那么，如果你已经把它们的表示层和结构分开了的话，就可以很容易地修改样式了。

```
.warning {
  font-weight: bold;
  color: blue;
}
```

如果把这个样式混杂在那 100 个文档里，则不得不进行大量的“搜索并替换”操作。

显然，把 CSS 代码从 HTML 文档里分离出来可以让 CSS 工作得最好。这个适用于 CSS 表示层的结论同样适用于 JavaScript 行为层。

5.4　分离 JavaScript

你们此前见到的JavaScript代码都已经与HTML文档分得很开了——负责实际完成各项任务的 JavaScript 函数都已存入有关的外部文件，而问题就出现在了内嵌的事件处理函数中。

类似于使用 style 属性，在 HTML 文档里使用诸如 onclick 之类的属性也是一种既没有效率又容易引发问题的做法。如果我们用一个“挂钩”，就像 CSS 机制中的 class 或 id 属性那样，把 JavaScript 代码调用行为与 HTML 文档的结构和内容分离开，网页就会健壮得多。那么，可否用下面这条语句来表明“当这个链接被点击时，它将调用 popUp()函数”的意思呢？

```
<a href="http://www.example.com/" class="popup">Example</a>
```

我很高兴告诉大家：完全可以这样做。JavaScript 语言不要求事件必须在 HTML 文档里处理，我们可以在外部 JavaScript 文件里把一个事件添加到 HTML 文档中的某个元素上：

```
element.event = action...
```

关键是怎样才能把应该获得这个事件的元素确定下来。这个问题可以利用 class 或 id 属性来解决。

如果你想把一个事件添加到某个带有特定 id 属性的元素上，用 getElementById()方法就可以解决问题：

```
getElementById(id).event = action
```

如果事情涉及多个元素，我们可以用 getElementsByTagName()和 getAttribute()方法把事件添加到有着特定属性的一组元素上。具体步骤如下（以 onclick 事件和 popUp()函数为例）：

(1) 把文档里的所有链接全放入一个数组里。

(2) 遍历数组。

(3) 如果某个链接的class属性等于popup，就说明这个链接在被点击时将调用popUp()函数。于是：

A. 把这个链接的 href 属性值传递给 popUp()函数。

B. 取消这个链接的默认行为，不让这个链接把访问者带离当前窗口。

下面是实现上述步骤的 JavaScript 代码：

```
var links = document.getElementsByTagName("a");
for (var i=0; i<links.length; i++) {
  if (links[i].className == "popup") {
    links[i].onclick = function() {
      popUp(this.getAttribute("href"));
      return false;
    }
  }
}
```

以上代码将把调用 popUp()函数的 onclick 事件添加到有关的链接上。只要把它们存入一个外部 JavaScript 文件，我们就等于是把这些操作从 HTML 文档里分离出来了。而这就是“分离 JavaScript”的含义。

还有个问题需要解决：如果把这段代码存入外部 JavaScript 文件，它们将无法正常运行。因为这段代码的第一行是：

```
var links = document.getElementsByTagName("a");
```

这条语句将在 JavaScript 文件被加载时立刻执行。因为此 JavaScript 文件是从 HTML 文档的<head>部分用<script>标签调用的，所以它将在 HTML 文档之前加载到浏览器里，而此时 HTML 文档还没有全部加载到浏览器里，文档模型也不完整。没有完整的 DOM，getElementsByTagName()等方法就不能正常工作。

必须让这些代码在 HTML 文档全部加载到浏览器之后才开始执行。还好，HTML 文档全部加载完毕时将触发一个事件，这个事件有它自己的事件处理函数。

HTML 文档将被加载到一个浏览器窗口里，document 对象又是 window 对象的一个属性。当 window 对象触发 onload 事件时，document 对象已经存在。

我将把我的 JavaScript 代码打包在 prepareLinks()函数里，并把这个函数添加到 window 对象的 onload 事件上去。这样一来，就可以正常工作了：

```
window.onload = prepareLinks;
function prepareLinks() {
  var links = document.getElementsByTagName("a");
  for (var i=0; i<links.length; i++) {
    if (links[i].getAttribute("class") == "popup") {
      links[i].onclick = function() {
        popUp(this.getAttribute("href"));
        return false;
      }
    }
  }
}
```

别忘记把 popUp()函数也保存到那个外部 JavaScript 文件里去：

```
function popUp(winURL) {
  window.open(winURL,"popup","width=320,height=480");
}
```

这是一个非常简单的例子，但它演示了怎样才能成功地把行为与结构分离开来。在本章稍后的内容里，我还会再向大家介绍几种可以在文档加载时把事件添加到有关元素上去的巧妙办法。

5.5 向后兼容性

正如前面反复强调的那样，你的网站的访问者很可能未激活 JavaScript 功能。同时，不同的浏览器对 JavaScript 的支持程度也是不一样的。

绝大多数浏览器都能或多或少地支持 JavaScript，而绝大多数现代的浏览器对 DOM 的支持都非常不错。但比较古老的浏览器却很可能无法理解 DOM 提供的方法和属性。

因此，即使某位用户在访问你的网站时使用的是支持 JavaScript 的浏览器，某些脚本也不一定能正常工作。

针对这一问题的最简单的解决方案是，在脚本里对浏览器对 JavaScript 的支持程度进行查询。这有点儿像游乐园里的警告牌："你必须达到这一身高才能参与这项游乐活动"。换句话说，需要在 DOM 脚本里表达出下面这个含义："你必须理解这么多的 JavaScript 语言才能执行这些语句"。

这个解决方案很容易实现：只要把某个方法打包在一个 if 语句里，就可以根据这条 if 语句的条件表达式的求值结果是 true（这个方法存在）还是 false（这个方法不存在）来决定应该采取怎样的行动。这种检测被称为对象检测（object detection）。在 JavaScript 语言里，几乎所有的东西（包括各种方法在内）都可以被当作对象来对待，而这意味着我们可以容易地把不支持某个特定 DOM 方法的浏览器检测出来：

```
if (method) {
statements
}
```

例如，如果有一个使用了 getElementById()方法的函数，就可以在调用 getElementById()方法之前先检查用户所使用的浏览器是否支持这个方法。因为我是想检查这个方法是否存在而不是想调用它，所以只需写出这个方法的名字（即 getElementById），而不需要（也不应该）像往常那样写出方法名后面的圆括号，如下所示：

```
function myFunction() {
  if (document.getElementById) {
    statements using getElementById
  }
}
```

因此，如果某个浏览器不支持 getElementById()方法，它就永远也不会执行使用此方法的语句。

这个解决方案的唯一不足是，如此编写出来的函数会增加一对花括号。如果需要在某个函数里对多个 DOM 方法和/或属性是否存在进行检测，这个函数中最重要的语句就会被深埋在一层又一层的花括号里。而这样的代码往往很难阅读和理解，而且很容易出现花括号不配对的问题。

把测试条件改为"如果你不理解这个方法，请离开"则更简单。

为了把测试条件从"如果你理解……"改为"如果你不理解……"，需要使用"逻辑非"操作符；这个操作符在 JavaScript 语言里被表示为一个惊叹号：

```
if (!method)
```

测试条件中的“……请离开”可以用一条 return 语句来实现。因为这相当于中途退出函数，所以让它返回布尔值 false 比较贴切。用来测试 getElementById()方法是否存在的语句如下所示：

```
if (!getElementById) {
  return false;
}
```

因为花括号部分只有 return false 一条语句，我们可以把它简写成一行：

```
if (!getElementById) return false;
```

如果需要测试多个方法或属性是否存在，可以用“逻辑或”操作符将其合并；这个操作符在 JavaScript 语言里被表示为两个竖线符号。如下所示：

```
if (!getElementById || !getElementsByTagName) return false;
```

如果这是游乐园里的一块警告牌的话，它的意思是“如果你不理解 getElementById 和 getElementsByTagName，你就不能参与这项游乐活动”。

现在，我将按照这一思路，在用来把 onclick 事件添加到一些链接上去的网页加载脚本里插入一条 if 语句。我在那个脚本里使用了 getElementsByTagName()方法，所以需要插入一条 if 语句（见黑体字部分）去检查浏览器是否理解这个方法：

```
window.onload = function() {
  if (!document.getElementsByTagName) return false;
  var lnks = document.getElementsByTagName("a");
  for (var i=0; i<lnks.length; i++) {
    if (lnks[i].getAttribute("class") == "popup") {
      lnks[i].onclick = function() {
        popUp(this.getAttribute("href"));
        return false;
      }
    }
  }
}
```

虽然只是一条简单的 if 语句，但它可以确保那些“古老的”浏览器不会因为我的脚本代码而出问题。这么做是为了让我的脚本有良好的向后兼容性。因为我在给网页添加各有关行为时始终遵循了“循序渐进”的原则，所以可以确实地知道我添加的那些功都有预留退路，我的网页在那些“古老的”浏览器里也能正常浏览。那些只支持一部分 JavaScript 功能、但不支持 DOM 的浏览器仍可以访问我的网页的内容。

浏览器嗅探技术

在 JavaScript 脚本代码里，在使用某个特定的方法或属性之前，先测试它是否真实存在是确保向后兼容性最安全和最可信的办法，但它并不是唯一的办法。在浏览器市场群雄逐鹿的那个年

代，一种称为浏览器嗅探（browser sniffung）的技术曾经非常流行。

“浏览器嗅探”技术是一套通过提取由浏览器供应商提供的信息来解决向后兼容问题的办法。从理论上讲，可以通过 JavaScript 代码检索关于浏览器品牌和版本的信息，这些信息可以用来改善 JavaScript 脚本代码的向后兼容性，但这是一种风险非常大的技术。

首先，浏览器有时会“撒谎”。因为历史原因，有些浏览器会把自己报告为另外一种用户代理，还有一些浏览器允许用户对这些信息进行任意修改。

其次，为了适用于多种不同的浏览器，浏览器嗅探脚本会变得越来越复杂。如果想让你的浏览器嗅探脚本能够跨平台工作，就必须对所有可能出现的供应商和版本号组合进行测试。这是一个无穷尽的任务，测试的组合情况越多，代码就越复杂和冗长。

最后，许多浏览器嗅探脚本在进行这类测试时要求浏览器的版本号必须得到精确的匹配。因此，每当市场上出现了新的浏览器或一个现有浏览器的新版本，就不得不修改这些脚本。

令人感到欣慰的是，充满着风险的浏览器嗅探技术正在被更简单也更健壮的对象检测技术所取代。

5.6 小结

在这一章里，我们向大家介绍了一些与 DOM 脚本编程工作有关的基本原则和良好习惯，它们是：

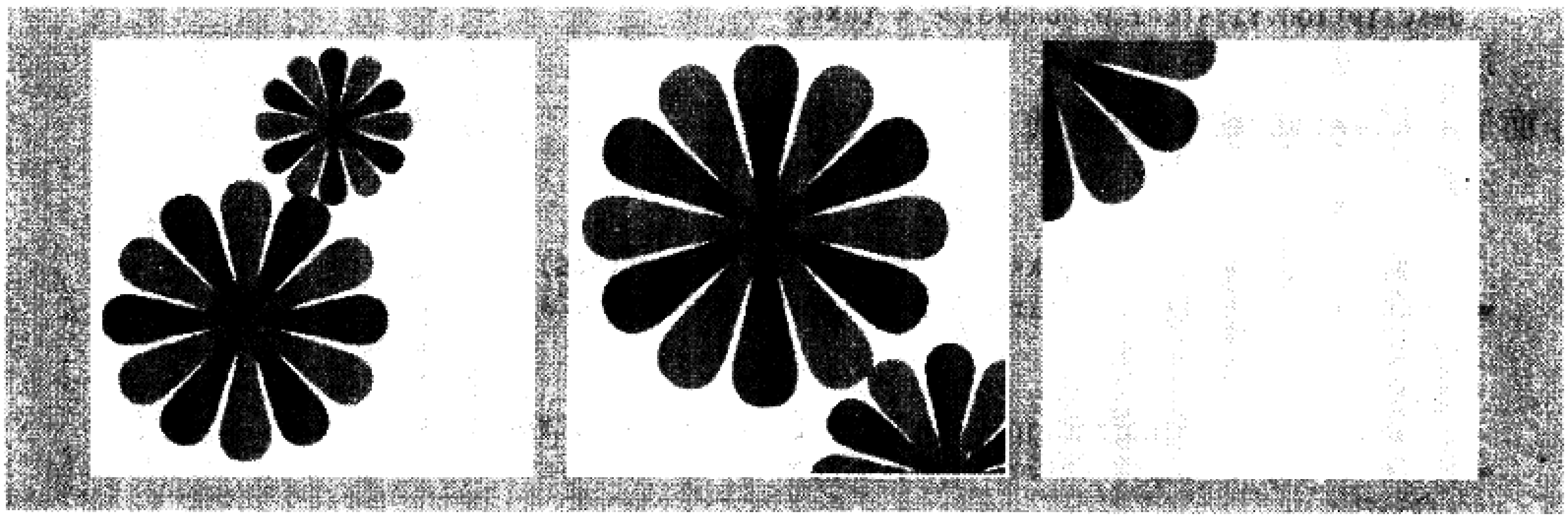
- ❑ 预留退路
- ❑ 分离 JavaScript
- ❑ 向后兼容性

本章的另一个主题是：在学习和使用 Flash 和 CSS 等其他一些技术时获得的经验可以帮助我们学好和用好 JavaScript。只有勤于思考、善于借鉴，才能编写出高品质的脚本。

在下一章里，我将以“勤于思考、善于借鉴”的态度对我的“JavaScript 美术馆”网页进行优化。我将重新检视它，找出并弥补它的不足和缺陷。

第6章

案例研究：JavaScript美术馆改进版



本章内容

- 把事件处理函数移出 HTML 文档
- 如何改善向后兼容性
- 确保可访问性

在第4章里，我们创建了一个“JavaScript美术馆”。在第5章里，我向大家介绍了一些JavaScript编程原则和良好习惯。在这一章里，我将运用那些原则和习惯去改进我的 JavaScript 美术馆。

“勤于思考，善于借鉴”是每位有创新精神的网页设计人员都应该具备的特质。无论是在编写 CSS 脚本还是在编写 JavaScript 脚本，也无论是直接编写代码还是使用可视化设计工具，一名优秀的网页设计人员总是会在每个细节上问自己这样一个问题：“是否还有更好的解决办法？”

正如在上一章里看到的那样，与 DOM 脚本编程工作有关的问题不外乎预留退路、向后兼容性和分离 JavaScript 这几大类。能否发现和解决这些问题对网页的可用性和可访问性有着决定性的影响。

6.1 快速回顾

在第 4 章里，我编写了一个用来替换“占位符”图片的 src 属性的脚本，那个脚本使我只用一个网页就建立起了 JavaScript 美术馆。下面是在第 4 章最终完成的图片切换函数 showPic()的代码清单：

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
  var text = whichpic.getAttribute("title");
  var description = document.getElementById("description");
  description.firstChild.nodeValue = text;
}
```

下面是用来调用 showPic()函数的 XHTML 文件片段：

```
<ul>
<li>
    <a href="images/fireworks.jpg" onclick="showPic(this);
➥return false;" title="A fireworks display">Fireworks</a>
  </li>
  <li>
    <a href="images/coffee.jpg" onclick="showPic(this); return false;"
➥title="A cup of black coffee">Coffee</a>
  </li>
  <li>
    <a href="images/rose.jpg" onclick="showPic(this); return false;"
➥title="A red, red rose">Rose</a>
  </li>
  <li>
    <a href="images/bigben.jpg" onclick="showPic(this); return false;"
➥title="The famous clock">Big Ben</a>
  </li>
</ul>
<p id="description">Choose an image.</p>
<img id="placeholder" src="images/placeholder.gif"
➥alt="my image gallery" />
```

现在，我将通过下面一些问题对这个解决方案进行改进。

6.2 解决“预留退路”问题

第一个问题是：“如果 JavaScript 功能被禁用，会怎样？”

仔细检查过 showPic()函数的代码后，我得出的结论是我的脚本已经为此预留了退路：即使 JavaScript 功能已被禁用，用户也可以毫无问题地浏览到 JavaScript 美术馆里的所有图片，网页里的所有链接也都可以正常工作：

```
<li>
  <a href="images/fireworks.jpg" onclick="showPic(this);
➥return false;" title="A fireworks display">Fireworks</a>
</li>
```

在没有 JavaScript“干扰”的情况下，浏览器将沿着 href 属性给出的链接前进，用户将看到一张新图片而不是“该页无法显示”之类的出错信息。虽说这与 JavaScript 在当前页面上切换显示新图片的效果要略差一些，但网页的基本功能并未受到损害——页面上的所有内容都可以访问。

如果我当初选用的是“javascript:”伪协议，链接将如下所示：

```
<li>
  <a href="javascript:showPic(' images/coffee.jpg'); return false;"
➥title="A cup of black coffee">Coffee</a>
</li>
```

如果我把这些链接都写成上面这样，它们在不支持或禁用了 JavaScript 功能的浏览器里将毫无用处。

类似地，把这些链接写成“#”记号也会导致类似的问题，但令人遗憾的是，这个技巧在那些利用剪贴操作“编写”的 JavaScript 代码里相当常见。类似于使用“javascript:”伪协议时的情况，如果我当初使用的是“#”记号，那些没有激活 JavaScript 功能的用户也将无法正常浏览我的 JavaScript 美术馆：

```
<li>
  <a href="#" onclick="showPic(' images/rose.jpg'); return false;"
➥title="A red, red rose">Rose</a>
</li>
```

把 href 属性设置为一个真实存在的值不过是举手之劳，但我的 JavaScript 美术馆却因此有了预留退路。虽说没有激活 JavaScript 功能的用户需要在浏览器里点击“后退”按钮才能重新看到我的图片清单，但这总比根本看不到那些图片要好得多吧。

我的 JavaScript 美术馆通过了第一个测试。

6.3 解决“分离 JavaScript”问题

下一个问题与在 HTML 文档里调用 JavaScript 代码的具体做法有关，它是这样的：HTML 文档的结构与 JavaScript 代码所实现的行为分开了吗？换句话说，网页的行为层（JavaScript）与其结构（XHTML）是彼此互不干扰、还是两种代码混杂在一起？

具体到 JavaScript 美术馆这个例子，这个问题的答案当然是“它们混杂在一起了”。

当初我是把 onclick 事件处理函数直接插入到 HTML 文档里的，如下所示：

```
<li>
  <a href="images/bigben.jpg" onclick="showPic(this); return false;"
➥title="The famous clock">Big Ben</a>
</li>
```

这显然不符合“分离 JavaScript”原则。我应该在外部文件里完成添加 onclick 事件处理函数的工作，那样才能让我的 HTML 文档没有“杂质”，就像下面这样：

```
<li>
  <a href="images/bigben.jpg" title="The famous clock">Big Ben</a>
</li>
```

把 JavaScript 代码移出 HTML 文档不是难事，但为了让浏览器知道页面里都有哪些链接有着“不正常”的行为，我必须找到一种“挂钩”把 JavaScript 代码与 HTML 文档中的有关标记关联起来。有多种办法可以让我达到这一目的。

我可以像下面这样给图片清单里的每个链接分别添加一个如下所示的 class 属性：

```
<li>
  <a href="images/bigben.jpg" class="gallerypic"
➥title="The famous clock">Big Ben</a>
</li>
```

但这个技巧还不够理想：给每个链接分别添加 class 属性与给它们分别添加事件处理函数同样麻烦。

图片清单里的各个链接有一个共同点：它们都包含在同一个列表清单元素（<ul>）里。给整个清单设置一个独一无二的 ID 的办法要简单得多：

```
<ul id="imagegallery">
<li>
    <a href="images/fireworks.jpg" title="A fireworks display">
➥Fireworks</a>
  </li>
  <li>
    <a href="images/coffee.jpg" title="A cup of black coffee">
➥Coffee</a>
  </li>
  <li>
    <a href="images/rose.jpg" title="A red, red rose">Rose</a>
  </li>
  <li>
    <a href="images/bigben.jpg" title="The famous clock">Big Ben</a>
  </li>
</ul>
```

正如你们将要看到的那样，虽然只有这一个“挂钩”，但对 JavaScript 来说已经足够了。

6.3.1 添加事件处理函数

现在，我需要编写一个简短的函数把有关操作关联到 onclick 事件上。我将其命名为 prepareGallery。

下面是我想让这个函数完成的工作：

- 检查当前浏览器是否理解 getElementsByTagName()方法。
- 检查当前浏览器是否理解 getElementById()方法。
- 检查当前网页是否包含着一个 id 属性值是"imagegallery"的元素。
- 构造一个循环来对"imagegallery"元素中的链接进行遍历处理。
- 对 onclick 事件处理函数进行设置，让它在有关链接被点击时完成以下操作：
  - 把这个链接作为参数传递给 showPic()函数。
  - 取消链接被点击时的默认行为，不让浏览器打开这个链接。

我将从定义 prepareGallery()函数开始。这个函数不需要参数，所以在这个函数名字后面的圆括号里用不着写出任何东西：

```
function prepareGallery() {
```

6.3.2 进行必要的检查

我想做的第一件事是检查当前浏览器是否理解名为 getElementsByTagName 的 DOM 方法。我将在这个函数里使用这个方法，我需要保证不理解这个方法的老浏览器不会执行这个函数：

```
if (!document.getElementsByTagName) return false;
```

这条 if 语句相当于这样一句话："如果 getElementsByTagName 未定义，请现在就离开。"理解这个 DOM 方法的浏览器将继续执行。

现在，对名为 getElementById 的 DOM 方法进行同样的检查，因为我的函数也会用到这个方法：

```
if (!document.getElementById) return false;
```

我可以把这两项检查组合在一起："只要你不理解这两个方法中的其中一个，请立刻离开"：

```
if (!document.getElementsByTagName || !document.getElementById)
➥return false;
```

不过，上面这样的代码开始变得比较冗长并难以阅读。事实上，从可读性的角度看，把多项测试写在同一行上的做法不一定是最好的主意。有不少程序员喜欢像下面这样把 return 语句单独写在一行上：

```
if (!document.getElementsByTagName)
  return false;
if (!document.getElementById)
  return false;
```

如果你也喜欢这样的写法，我建议你最好是用花括号把 return 语句括起来，如下所示：

```
if (!document.getElementsByTagName) {
  return false;
}
if (!document.getElementById) {
```

```
    return false;
  }
```

这或许是最简明、最具有可读性的代码书写方式。

把这些测试写在同一行上还是写成好几行是你们的自由，你们完全可以根据自己的偏好来做出选择。

完成这两项具有普遍适用性的测试后，我还安排了一个更具针对性的测试。我正在编写的这个函数将处理 id 属性值等于 imagegallery 的那个元素所包含的链接，假如这个元素并不存在，我的这个函数也就无需继续执行了。

与前面两项测试一样，我将使用“逻辑非”操作符来进行这一测试：

```
if (!document.getElementById("imagegallery")) return false;
```

出于个人偏好，你们也许更喜欢下面这样的写法：

```
if (!document.getElementById("imagegallery")) {
  return false;
}
```

这项测试是一个预防性措施。我知道我的 HTML 文档——它将调用我正在编写的这个 JavaScript 函数，里有一个 id 属性值等于 imagegallery 的列表清单元素，但我不敢确定这在将来会不会发生变化。有了这个预防性措施，即使以后因为某种原因我决定从网页上删掉 JavaScript 美术馆，我也用不着担心这个网页的 JavaScript 代码会突然变得漏洞百出。把 HTML 文档的内容与 JavaScript 代码所实现的操作行为分离开来的重要性和好处由此可见一斑。作为一条原则，如果你想用 JavaScript 给某个网页添加一些行为，就不应该让你的 JavaScript 代码对这个网页的结构有任何依赖。

结构化程序设计原则的灵活运用

结构化程序设计（structed programming）理论提出了这样一条原则：每个函数应该只有一个入口点和一个出口点。

我刚才的做法已经违背了这一原则：我在 prepareGallery()函数的开头部分使用了多条 return false 语句，它们全都是这个函数的出口点。根据结构化程序设计理论的有关原则，应该把这些出口点减少到一个。

从理论上讲，我很赞同这项原则；但在实际工作中，过分拘泥于这项原则往往会使代码变得非常难以阅读。如果为了避免在 prepareGallery()函数里留下多个出口点而去改写那些 if 语句的话，这个函数的核心代码就会被掩埋在一层又一层的花括号里，就像下面这样：

```
function prepareGallery() {
  if (document.getElementsByTagName) {
    if (document.getElementById) {
      if (document.getElementById("imagegallery")) {
```

```
          statements go here...
        }
      }
    }
  }
```

我个人认为，同一个函数有多个出口点的情况是可以接受的，但前提是它们应该集中出现在这个函数的开头部分。

出于可读性的考虑，我将把那些 return false 语句全部集中到 prepareGallery()函数的开头部分：

```
function prepareGallery() {
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("imagegallery")) return false;
```

把必要的测试和检查工作都安排就绪之后，我把目光转向了 prepareGallery()函数的核心功能。

6.3.3 创建必要的变量

首先，我想把事情弄得稍微简单些：重复多次地敲入像 document.getElementById("imagegallery")这么长的字符串实在很麻烦，所以我决定创建一个名为 gallery 的变量来简化之：

```
var gallery = document.getElementById("imagegallery");
```

在不违反 JavaScript 变量命名规则的前提下，完全可以给变量起一个另外的名字。我之所以选用“gallery”来命名这个变量，是因为它的含义就是“美术馆”。选择一些有意义的单词来命名你的变量可以让代码更容易阅读和理解。

注意 在为变量命名时一定要谨慎从事。有些单词在JavaScript语言里有特殊的含义和用途，这些被统称为“保留字”的单词不能用做变量名。另外，现有JavaScript函数或方法的名字也不能用来命名变量。不要使用诸如alter、var或if之类的单词作为变量的名字。

按照计划，我需要在 prepareGallery()函数里构造一个循环去遍历 imagegallery 元素中的所有链接，在那个循环里我会用到 getElementsByTagName()方法。利用刚刚创建的 gallery 变量，我可以把对 getElementsByTagName()方法的调用简单地写成下面这个样子：

```
gallery.getElementsByTagName("a")
```

它等价于下面这个长长的记号：

```
document.getElementById("imagegallery").getElementsByTagName("a")
```

现在，我想把事情弄得更简单些。我决定把上面这个记号所代表的数组（更准确地说，是一个节点列表）赋值给一个变量，并将该变量命名为“links”：

```
var links = gallery.getElementsByTagName("a");
```

下面是 prepareGallery()函数截至到目前的样子：

```
function prepareGallery() {
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("imagegallery")) return false;
  var gallery = document.getElementById("imagegallery");
  var links = gallery.getElementsByTagName("a");
```

准备工作已就绪。我已经安排好了必要的测试和检查工作，还创建了几个变量。

6.3.4 创建循环

我想遍历 links 数组里的各个元素。我将使用一个 for 循环来完成这项工作。

首先，需要一个变量来充当这个循环的循环计数器并把它的初始值设置为零。在这个循环里每遍历 links 数组里的一个元素，这个计数器就将增加一个 1。下面是对这个计数器进行初始化的语句：

```
var i = 0;
```

把充当循环计数器的变量命名为“i”是一种传统做法。字母“i”在这里的含义是“increment”（递增），许多程序设计语言都使用“i”作为变量值按顺序递增的变量的名字。

下面是这个循环的控制条件：

```
i  < links.length;
```

上面这个表达式的含义是：只要变量 i 的值小于 links 数组的 length 属性值，这个 for 循环就一直循环下去。在这里，length 属性的值等于 links 数组里的元素总个数。因此，如果 links 数组包含着 4 个元素，那么只要 i 小于 4，我的 for 循环就将一直循环下去。

最后，使用下面这个记号来给循环计数器加上一个 1：

```
i++;
```

这个记号是下面这条语句的简写形式：

```
i = i+1;
```

上面这几条语句的整体效果是：这个 for 循环每执行一次，变量 i 的值就会加 1；一旦变量 i 的值不再小于 links.length，循环就结束。因此，如果 links 数组包含着 4 个元素，这个循环将在 i 等于 4 时结束执行。这个循环将总共执行 4 次——别忘了，变量 i 是从零开始计数的。

下面是 for 循环的开头部分：

```
for ( var i=0; i < links.length; i++) {
```

6.3.5 完成必要的操作

具体到这个例子，我想要完成的操作是改变 links 数组中的各个元素的行为。根据它所包含的元素，与其说 links 是一个数组，不如说它是一个节点列表（node list）来得更准确。它是一个由 DOM 节点构成的集合，这个集合里的每个节点都有自己的属性和方法。

在这个例子里我最感兴趣的是 onclick 方法。我将利用如下所示的语法把一种行为赋予这个方法：

```
links[i].onclick = function() {
```

这条语句定义了一个匿名函数。这是一种“临时抱佛脚”式的函数创建办法，最适合用来定义在整个脚本里只出现一次的函数：匿名函数没有名字，只能在哪里定义、在哪里使用。具体到上面这条语句，它将在 links[i]元素的 onclick 事件处理函数被触发时创建一个匿名函数。我将要放到这个匿名函数里去的所有语句将在 links[i]元素所对应的链接被点击时得到执行。

links[i]元素的值会随着变量 i 的递增而变化。如果假设 links 集合里包含着 4 个元素，那么第一个元素将是 links[0]，最后一个元素是 links[3]。

我将传递给 showPic()函数的参数是关键字 this，它代表着在此时此刻与 onclick 方法相关联的那个元素。也就是说，this 在这里代表着 links[i]，而 links[i]又对应着 links 节点列表里的某个特定的节点：

```
showPic(this);
```

我还需要再多做一件事，即禁用有关链接的默认行为。我的想法是：如果 showPic()函数执行成功，就不让浏览器去执行某个链接被点击时的默认操作。和以前一样，我想取消这种默认行为，不让浏览器前进到那个链接所指向的目的地：

```
return false;
```

返回布尔值 false 相当于向浏览器传递了这样一条消息：“按照这个链接没被点击的情况采取行动。”

最后，还需要用一个右花括号来结束这个匿名函数。下面是我最终完成的匿名函数：

```
links[i].onclick = function() {
  showPic(this);
  return false;
}
```

6.3.6 完成 JavaScript 函数

现在，需要用一个右花括号来结束 for 循环：

```
for ( var i=0; i < links.length; i++) {
  links[i].onclick = function() {
    showPic(this);
    return false;
  }
}
```

最后，只要再用一个右花括号来结束 prepareGallery()函数。下面是这个函数完整的代码清单：

```
function prepareGallery() {
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("imagegallery")) return false;
  var gallery = document.getElementById("imagegallery");
  var links = gallery.getElementsByTagName("a");
  for ( var i=0; i < links.length; i++) {
    links[i].onclick = function() {
      showPic(this);
      return false;
    }
  }
}
```

调用此函数时，它将把 onclick 事件绑定到 id 属性值等于“imagegallery”的那个元素所包含的各个链接上去。

6.3.7　把多个 JavaScript 函数绑定到 onload 事件处理函数上

我必须执行 prepareGallery()函数才能对 onclick 事件进行绑定。

如果现在就执行这个函数，它将无法完成其工作：在 HTML 文档完成加载之前，DOM 是不完整的。具体到 prepareGallery()函数，它的第 3 行代码将测试"imagegallery"元素是否存在，如果 DOM 不完整，这项测试的准确性就无从谈起，事态的发展就会偏离我的计划。

必须让这个函数只在网页加载完毕之后才得到执行。网页加载完毕时会触发一个 onload 事件，这个事件与 window 对象相关联。为了让事态的发展不偏离计划，必须把 prepareGallery()函数绑定到这个事件上：

```
window.onload = prepareGallery;
```

这条浅显易懂的语句解决了我的问题，但它现在的样子还有点儿短视。

假设我有两个函数：firstFunction()和 secondFunction()。如果我想让它们俩都在页面加载时得到执行，我该怎么办？如果把它们逐一绑定到 onload 事件上，它们当中将只有最后那个才会被实际执行：

```
window.onload = firstFunction;
window.onload = secondFunction;
```

secondFunction 将取代 firstFunction。由此我们可以得出一个结论：每个事件处理函数只能绑定一条指令。

还好，这里有个小技巧可以让我绕过这一难题：可以先创建一个匿名函数来容纳这两个函数，然后把那个匿名函数绑定到 onload 事件上，如下所示：

```
window.onload = function() {
  firstFunction();
  secondFunction();
}
```

这个技巧既简明又实用——在需要绑定的函数不是很多的场合，这应该是最简明的解决方案了。

还有一个更通用的解决方案——不管你打算在页面加载完毕时执行多少个函数，它都可以应付自如。这个更通用的解决方案需要额外编写一些代码，但好处是一旦有了那些代码，把函数，不管它们有多少，绑定到 window.onload 事件上的工作就非常简明易行了。

这个函数的名字是 addLoadEvent，它是由 Simon Willison（详见 http://simon.incutio.com）编写的。它只有一个参数：打算在页面加载完毕时执行的函数的名字。

下面是 addLoadEvent()函数将要完成的操作：

- 把现有的 window.onload 事件处理函数的值存入变量 oldonload。
- 如果在这个处理函数上还没有绑定任何函数，就像平时那样把新函数添加给它。
- 如果在这个处理函数上已经绑定了一些函数，就把新函数追加到现有指令的末尾。

下面是 addLoadEvent()函数的代码清单：

```
function addLoadEvent(func) {
  var oldonload = window.onload;
  if (typeof window.onload != 'function') {
    window.onload = func;
  } else {
    window.onload = function() {
      oldonload();
      func();
    }
  }
}
```

这相当于把那些将在页面加载完毕时执行的函数创建为一个队列。如果想把刚才那两个函数添加到这个队列里去，只需写出以下代码就行了：

```
addLoadEvent(firstFunction);
addLoadEvent(secondFunction);
```

我发现这个函数非常实用，尤其是在代码变得越来越复杂的时候。无论我打算在页面加载完毕时执行多少个函数，都只要多写一条语句就可以安排好一切。

这个解决方案对 prepareGallery()函数来说好像有点儿大材小用，因为只有这一个函数需要在页面加载完毕时执行。可是，为以后的扩展做一些准备工作总不是件坏事。我决定把 addLoadEvent()函数收录到我的脚本里，这使我只需写出下面这行代码就可以把 prepareGallery()函数绑定到 onload 事件上：

```
addLoadEvent(prepareGallery);
```

经过努力，prepareGallery()函数已经没有任何毛病了——至少我是这么认为的。接下来，我将把目光转向在第 4 章编写出来的 showPic()函数。

6.4 JavaScript 函数的优化：不要做太多的假设

我在 showPic()函数里发现的第一个问题是，我没有让它进行任何测试和检查。

showPic()函数将由 prepareGallery()函数调用，而我已经在 prepareGallery()函数的开头对 getElementById()和 getElementsByTagName()等 DOM 方法是否存在进行过检查，所以我确切地知道用户的浏览器不会因为不理解这两个方法而出问题。

不过，我还是在 showPic()函数里做出了太多的假设。别的先不说，我在 showPic()函数的代码里用到了 id 属性值等于 palceholder 和 description 的元素，但我并未对这些元素是否存在做任何检查：

```
function showPic(whichpic) {
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
  var text = whichpic.getAttribute("title");
  var description = document.getElementById("description");
  description.firstChild.nodeValue = text;
}
```

我需要增加一些语句来检查这些元素是否存在。

showPic()函数负责完成两件事：一是找出 id 属性值是 palceholder 的图片并修改其 src 属性；二是找出 id 属性是 description 的元素并修改其第一个子元素（firstChild）的 nodeValue 属性。第一件事是这个函数必须完成的任务，第二件事只是一项多多益善的补充。因此，我决定把检查工作分成两个步骤以获得这样一种效果：只要 palceholder 图片存在，即使 description 元素不存在，切换显示新图片的操作也将照常进行。

正如你们在 prepareGallery()函数里看到的那样，检查某个特定的元素是否存在是一件很简单的事情：

```
if (!document.getElementById("placeholder")) return false;
```

紧随其后的是用来修改 palceholder 图片的 src 属性的代码，它们的效果是切换显示一张新图片：

```
var source = whichpic.getAttribute("href");
var placeholder = document.getElementById("placeholder");
placeholder.setAttribute("src",source);
```

以上代码负责完成 showPic()函数的主要任务。接下来，检查 description 元素是否存在：

```
if (!document.getElementById("description")) return false;
```

只有通过了这项检查，负责修改图片说明文字的代码才会得到执行，如下所示：

```
var text = whichpic.getAttribute("title");
var description = document.getElementById("description");
description.firstChild.nodeValue = text;
```

下面是 showPic()函数在我给它增加了两项检查（见黑体字部分）之后的代码清单：

```
function showPic(whichpic) {
  if (!document.getElementById("placeholder")) return false;
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
  if (!document.getElementById("description")) return false;
  var text = whichpic.getAttribute("title");
  var description = document.getElementById("description");
  description.firstChild.nodeValue = text;
}
```

改进后的 showPic()函数不再假设有关 HTML 文档里肯定存在着 palceholder 图片和 description 元素。即使 HTML 文档里没有 palceholder 图片，也不会发生任何 JavaScript 错误。

可是，还有一个问题没有得到解决：如果把 palceholder 图片从 HTML 文档里删掉并在浏览器里刷新这页面，就会出现无论点击 imagegellery 清单里的哪一个链接都不会有任何事情发生的情况。

这意味着我们的脚本没有足够的预留退路。此时，我们应该让浏览器打开那个被点击的链接而不是让什么事情都不发生。

问题的根源在于 prepareGallery()函数做出了这样一个假设：showPic()函数肯定会正常返回。基于这一假设，prepareGallery()函数取消了 onclick 事件的默认行为：

```
links[i].onclick = function() {
  showPic(this);
  return false;
}
```

是否要返回一个 false 值以取消 onclick 事件的默认行为，其实是一个应该由 showPic()函数而不是由 prepareGallery()函数来做出的决定。

虽然有点儿难以理解，但为了解决这个问题，我决定让 showPic()函数做以下事情：

- 如果图片切换成功，showPic()函数应该返回 false。
- 如果图片切换不成功，showPic()函数应该返回 true。

为了实现上述思路，我需要对 showPic()函数中的第一项检查做出这样的修改：如果 palceholder 图片不存在，showPic()函数应该返回 true 而不是 false：

```
if (!document.getElementById("placeholder")) return true;
```

showPic()函数中的第二项检查不需要修改。我的想法是：只要图片切换成功，即使 description 元素未被刷新，也要取消 onclick 事件的默认行为：

```
if (!document.getElementById("description")) return false;
```

作为实现上述思路的关键，还需要在 showPic()函数的末尾加上一条 return false 语句。因为执行到 showPic()函数的末尾意味着“切换图片”和“刷新 description 元素”这两项任务都取得了成功，此时必须取消 onclick 事件的默认行为：

```
return false;
```

现在，如果 showPic()函数没有成功，它将返回 true；如果这个函数完成了它的主要任务（切换图片），它将返回 false。下面是 showPic()函数在我完成上述修改后的代码清单：

```
function showPic(whichpic) {
  if (!document.getElementById("placeholder")) return true;
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
  if (!document.getElementById("description")) return false;
  var text = whichpic.getAttribute("title");
  var description = document.getElementById("description");
  description.firstChild.nodeValue = text;
  return false;
}
```

接下来，为了让 showPic()函数所返回的布尔值发挥作用，我还需要对 prepareGallery()函数做些修改：

```
links[i].onclick = function() {
  return showPic(this);
}
```

此时，如果 showPic()返回 false，prepareGallery()函数将把这个布尔值传递给浏览器，而这意味着 onclick 事件的默认行为不会发生——浏览器不会去打开那个链接。

反之，如果 showPic()函数返回的是 true，浏览器将打开那个链接。

下面是 prepareGallery()函数现在的代码清单：

```
function prepareGallery() {
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("imagegallery")) return false;
  var gallery = document.getElementById("imagegallery");
  var links = gallery.getElementsByTagName("a");
  for ( var i=0; i < links.length; i++) {
    links[i].onclick = function() {
      return showPic(this);
    }
  }
}
```

经过一番周折，终于把“JavaScript 美术馆”里的最后一个已知问题解决了：如果 palceholder 图片不存在，浏览器将沿着用户所点击的那个链接去打开一张新图片。现在可以把 palceholder 图片重新放回到 HTML 文档里去了。

6.4.1 不放过每个细节

showPic()和 prepareGallery()函数已经相当完善了。虽然它们的长度有所增加，但它们对 HTML 文档的依赖和假设已经比原先少多了。

尽管如此，在 showPic()函数里仍存在着一些假设。虽说它们不影响“JavaScript 美术馆”的基本功能，但我认为不应该放过每个细节。

比如说，showPic()函数假设每个链接都有一个 title 属性：

```
var text = whichpic.getAttribute("title");
```

为了检查 title 属性是否真的存在，我可以测试它是不是等于 null：

```
if (whichpic.getAttribute("title") != null)
```

如果 title 属性存在，这个 if 表达式将被求值为 true。如果 title 属性不存在，whichpic.getAttribute("title")将等于 null，而这个 if 表达式将被求值为 false。

这个 if 表达式还可以简写为：

```
if (whichpic.getAttribute("title"))
```

只要 title 属性存在，这个 if 表达式就将返回一个 true 值。

作为一种简单的视觉反馈，我决定在 title 属性不存在时把变量 text 的值设置为空字符串：

```
if (whichpic.getAttribute("title")) {
  var text = whichpic.getAttribute("title");
} else {
  var text = "";
}
```

下面是完成同样操作的另一种办法：

```
var text = whichpic.getAttribute("title") ?
➥whichPic.getAttribute("title") : "";
```

紧跟在 whichpic.getAttribute("title")后面的问号（?）是一个三元操作符（ternary operator）。这个问号的后面是变量 text 的两种可取值。如果 getAttribute("title")的返回值不是 null，text 变量将被赋值为第一个值；如果 getAttribute("title")的返回值是 null，text 变量将被赋值为第二个值：

```
variable = condition ? if true : if false;
```

如果 title 属性存在，变量 text 将被赋值为 whichpic.getAttribute("title")。如果 title 属性不存在，变量 text 将被赋值为一个空字符串（""）。

三元操作符是 if-else 语句的一种变体形式。它比较简短，但逻辑关系表达得不那么明显。如果你也这么认为，那就使用 if-else 语句好了。

现在，如果试图把 imagegallery 清单里的某个链接的 title 属性删掉并重新加载这个页面，当你再去点击那个链接的时候，description 元素将被填入一个空字符串。

如果想做到十全十美的话，可以对任何一种情况进行检查。

比如说，showPic()函数对 palceholder 元素是否存在进行了检查，但它需要假设那是一张图片。为了验证这种情况，我可以用 nodeName 属性来增加一项测试：

```
if (placeholder.nodeName != "IMG") return true;
```

请注意，nodeName 属性总是返回一个大写字母的值，即使元素在 HTML 文档里是小写字母。

我还可以引入更多的检查。比如说，showPic()函数需要假设 description 元素的第一个子元素（firstChild）是一个文本节点。我应该对此进行检查。

我可以利用 nodeType 属性来进行这项检查。还记得吗，文本节点的 nodeType 属性值等于 3：

```
if (description.firstChild.nodeType == 3) {
  description.firstChild.nodeValue = text;
}
```

下面是在我引入了以上几项检查之后 showPic()函数的代码清单：

```
function showPic(whichpic) {
  if (!document.getElementById("placeholder")) return true;
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  if (placeholder.nodeName != "IMG") return true;
  placeholder.setAttribute("src",source);
  if (!document.getElementById("description")) return false;
  var text = whichPic.getAttribute("title") ?
➥whichPic.getAttribute("title") : "";
```

```
    var description = document.getElementById("description");
    if (description.firstChild.nodeType == 3) {
      description.firstChild.nodeValue = text;
    }
    return false;
  }
```

因为又增加了几项检查，showPic()函数里的代码变得更多了。在实际工作中，这些检查很可能都是不必要的。它们针对的是 HTML 文档不在你们控制范围内的情况。如果你们有权对 HTML 文档进行修改，那就用不着这么麻烦。不过，作为一条原则，你们的脚本绝不应该对 HTML 文档的内容和结构做太多的假设。

这方面的决定需要根据具体情况来做出。

6.4.2 键盘浏览功能

在牵涉到 onclick 事件处理函数的脚本里，有一项优化工作是不能不考虑的。

下面是 prepareGallery()函数中的一段重要代码：

```
links[i].onclick = function() {
  return showPic(this);
}
```

这段代码本身没有任何毛病，其含义是：当这个链接被点击时，showPic()函数就开始执行。不过，这给人一种印象：用户只能使用鼠标来点击（click）这个链接。

但是，千万不要忘记并非所有的用户都使用鼠标来进行浏览。比如说，有视力残疾的用户往往无法看清屏幕上四处移动的鼠标指针。他们往往更喜欢使用键盘进行浏览。

作为一个众所周知的事实，不使用鼠标也可以浏览 Web。键盘上的 TAB 键可以让我们从这个链接移动到另一个链接，而按下回车键将激活当前链接。

有个名叫 onkeypress 的事件处理函数是专门用来处理键盘事件的。按下键盘上任何一个按键都会触发 onclick 事件。

如果想让 onkeypress 事件与 onclick 事件触发同样的行为，我可以简单地把有关指令复制一份：

```
links[i].onclick = function() {
  return showPic(this);
}
links[i].onkeypress = function() {
  return showPic(this);
}
```

还有一种更简单的办法可以确保 onkeypress 事件与 onclick 事件有着同样的行为：

```
links[i].onkeypress = links[i].onclick;
```

上面这条语句将把 onclick 事件的所有功能赋值给 onkeypress 事件：

```
links[i].onclick = function() {
  return showPic(this);
}
links[i].onkeypress = links[i].onclick;
```

请注意，我们之所以能这么做，是因为我们遵守了“分离 JavaScript”原则。

如果你已经把所有的函数和事件处理函数都放在了外部文件里，就可以在不影响 HTML 文档的情况下对其进行修改。你可以随时打开脚本并对它们进行优化，而你做出的修改将自动作用于每个引用了这个 JavaScript 文件的网页。

反之，如果你仍在使用内嵌于 HTML 文档的事件处理函数，在对 JavaScript 功能做出修改之后，你可能需要打开 HTML 文档去进行大量的修改。比如说，在“JavaScript 美术馆”这个例子里，我当初曾使用过一些如下所示的内嵌型事件处理函数：

```
<li>
  <a href="images/fireworks.jpg" onclick="showPic(this);
➥return false;" title="A fireworks display">Fireworks</a>
</li>
```

假如我没有把 onclick 事件处理函数从 HTML 文档里分离出来，在对 showPic()函数返回 true 或 false 的情况做出调整之后，我将不得不对 HTML 文档里的 onclick 事件处理函数做出相应的修改，如下所示：

```
<li>
  <a href="images/fireworks.jpg" onclick="return showPic(this);"
➥title="A fireworks display">Fireworks</a>
</li>
```

如果我的“JavaScript 美术馆”链接有大量图片的话，完成这种修改的工作量就会非常大。

在使用内嵌型事件处理函数的情况下，如果我想添加 onkeypress 事件处理函数，就不得不遍历所有的链接并给它们当中的每个再增加一个内嵌型事件处理函数：

```
<li>
  <a href="images/fireworks.jpg" onclick="return showPic(this);"
➥onkeypress="return showPic(this);"
➥title="A fireworks display">Fireworks</a>
</li>
```

这是一件非常麻烦又非常容易出错的事情。因为我已经把 JavaScript 代码从 HTML 文档分离到了外部文件里，所以我现在只需修改很少几条语句就可以把一切安排妥当。

6.4.3　慎用 onkeypress 事件处理函数

如你所见，我决定不添加 onkeypress 事件处理函数。这个事件处理函数很容易引起问题。用户每按下键盘上一个按键都会触发它。在某些浏览器里，那甚至包括 TAB 键！这意味着如果

绑定在 onkeypress 事件处理函数上的某个函数返回的是 false，那些使用键盘来进行浏览的用户就将无法离开当前链接。我的“JavaScript 美术馆”网页就存在着这样的问题——只要图片切换成功，showPic()函数就将返回 false。

怎么办？我可不想让那些使用键盘的用户无法浏览我的“JavaScript 美术馆”。

很幸运，onclick 事件处理函数比我们想像的更聪明。虽然它的名字“onclick”给人以一种它只与鼠标点击动作相关联的印象，但事实却并非如此：在几乎所有的浏览器里，用 TAB 键移动到某个链接然后按下回车键的动作也会触发 onclick 事件。从这一点来看，把它命名为“onactivate”也许更恰如其分。

围绕着 onclick 和 onkeypress 有许多让人困惑的东西，它们的名字是造成这些困惑的主要原因。有些教科书建议人们为每个 onclick 事件处理函数配上一个 onkeypress 事件处理函数。而事实上，因为这种搭配而导致的问题往往要比它们解决的更多。

如果无特殊理由，最好不要使用 onkeypress 事件处理函数。onclick 事件处理函数已经足以解决几乎所有的问题。不要因为这个事件处理函数的名字是“onclick”就产生误解，它对键盘浏览功能的支持相当完美。

下面是我最终完成的 prepareGallery()和 showPic()函数的代码清单：

```
function prepareGallery() {
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("imagegallery")) return false;
  var gallery = document.getElementById("imagegallery");
  var links = gallery.getElementsByTagName("a");
  for ( var i=0; i < links.length; i++) {
    links[i].onclick = function() {
      return showPic(this);
    }
  }
}
function showPic(whichpic) {
  if (!document.getElementById("placeholder")) return true;
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  if (placeholder.nodeName != "IMG") return true;
  placeholder.setAttribute("src",source);
  if (!document.getElementById("description")) return false;
  var text = whichpic.getAttribute("title") ?
➥whichpic.getAttribute("title") : "";
  var description = document.getElementById("description");
  if (description.firstChild.nodeType == 3) {
    description.firstChild.nodeValue = text;
  }
  return false;
}
```

可以通过本书在“Friends of ED”网站（http://www.friendsofed.com）上的主页来下载这两个函数的最终版本。

6.4.4　把 JavaScript 与 CSS 结合起来

把 JavaScript 代码从 HTML 文档里分离出去还可以带来另一个好处。在把内嵌型事件处理函数移出 HTML 文档时，我在 HTML 文档里为 JavaScript 代码留下了一个“挂钩”：

```
<ul id="imagegallery">
```

这个挂钩完全可以用在 CSS 样式表里。

比如说，如果我不想把图片清单显示成一个带项目符号的列表，则完全可以利用标识符 imagegallery 写出一条如下所示的 CSS 语句：

```
#imagegallery {
  list-style: none;
}
```

我可以把这条 CSS 语句存入一个外部文件——比如 layout.css 文件，然后再从 gallery.html 文件的<head>部分引用它：

```
<link rel="stylesheet" href="styles/layout.css" type="text/css"
➥ media="screen" />
```

利用 CSS，我甚至可以让这份清单里的列表项从按纵向显示变成按横向显示：

```
#imagegallery li {
  display: inline;
}
```

上面这两条 CSS 语句将使我的网页变成如图所示的样子。



即使把图片链接显示为一些缩微图而不是文字，CSS 也依然有效：

```
<ul id="imagegallery">
  <li>
    <a href="images/fireworks.jpg" title="A fireworks display">
      <img src="images/thumbnail_fireworks.jpg" alt="Fireworks" />
    </a>
  </li>
  <li>
    <a href="images/coffee.jpg" title="A cup of black coffee" >
      <img src="images/thumbnail_coffee.jpg" alt="Coffee" />
    </a>
  </li>
  <li>
    <a href="images/rose.jpg" title="A red, red rose">
      <img src="images/thumbnail_rose.jpg" alt="Rose" />
    </a>
  </li>
  <li>
    <a href="images/bigben.jpg" title="The famous clock">
      <img src="images/thumbnail_bigben.jpg" alt="Big Ben" />
    </a>
  </li>
</ul>
```

下面是这个网页的新形象，图片链接都被显示为缩略图而不是文本内容。

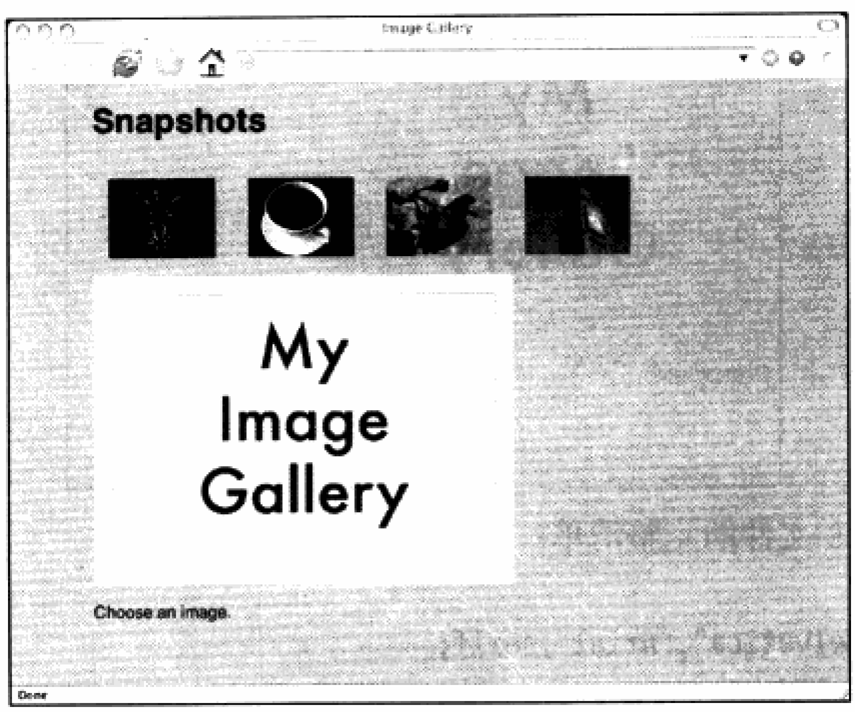


下面是 layout.css 文件的完整清单：

```
body {
  font-family: "Helvetica","Arial",serif;
  color: #333;
```

```
  background-color: #ccc;
  margin: 1em 10%;
}
h1 {
  color: #333;
  background-color: transparent;
}
a {
  color: #c60;
  background-color: transparent;
  font-weight: bold;
  text-decoration: none;
}
ul {
  padding: 0;
}
li {
  float: left;
  padding: 1em;
  list-style: none;
}
#imagegallery {
  list-style: none;
}
#imagegallery li {
  display: inline;
}
#imagegallery li a img {
  border: 0;
}
```

这份样式表将把我的“JavaScript 美术馆”页面装点得既美观又大方。



6.5 DOM Core 和 HTML-DOM

至此，我在编写 JavaScript 代码时只用到了以下几个 DOM 方法：

- getElementById()方法
- getElementsByTagName()方法
- getAttribute()方法
- setAttribute()方法

这些方法都是 DOM Core 的组成部分。它们并不专属于 JavaScript，支持 DOM 的任何一种程序设计语言都可以使用它们。它们的用途也并非仅限于处理网页，它们可以用来处理用任何一种标记语言（比如 XML）编写出来的文档。

在使用 JavaScript 语言和 DOM 为(X)HTML 文件编写脚本时，还有许多属性可供选用。这些属性专属于 HTML-DOM，而这一模型在 DOM Core 出现之前很久就已经为人们所熟悉了。

比如说，HTML-DOM 提供了一个 forms 对象。这个对象可以把下面这样的语句：

```
document.getElementsByTagName("form")
```

简化为：

```
document.forms
```

类似地，HTML-DOM 还提供了一些更简明的记号来描述各种 HTML 元素的属性。比如说，HTML-DOM 为图片提供的 src 属性可以把下面这样的语句：

```
element.getAttribute("src")
```

简化为：

```
element.src
```

这些方法和属性可以相互替换。同样的操作既可以只使用 DOM Core 来实现，也可以使用 HTML-DOM 来实现，最终的效果并无区别。正如大家看到的那样，HTML-DOM 记号通常比较简短，但它们只能用来处理 Web 文档——如果你们打算用 DOM 去处理其他类型的文档，请千万注意这一点。

如果我当初决定使用 HTML-DOM 来实现“JavaScript 美术馆”的话，就可以把 showPic() 函数里的多条语句写得简短一些。

比如说，下面这条语句使用了由 DOM Core 提供的 getAttribute()方法来检索 whichpic 元素的 href 属性，检索出来的 href 属性值被赋给了变量 source：

```
var source = whichpic.getAttribute("href");
```

下面是使用 HTML-DOM 提供的 href 属性来达到同样目的的语句：

```
var source = whichpic.href;
```

下面这条语句使用了由 DOM Core 提供的 setAttribute()方法，它将把 placeholder 元素的 src 属性设置为变量 source 的值：

```
placeholder.setAttribute("src",source);
```

下面是使用 HTML-DOM 提供的 src 属性来达到同样目的的语句：

```
placeholder.src = source ;
```

即使你们决定只使用 DOM Core 提供的方法来编写脚本，你们也应该去了解 HTML-DOM。在阅读别人编写的脚本源代码时难免会遇到各种 HTML-DOM 记号，你们至少应该知道那些记号都是干什么用的。

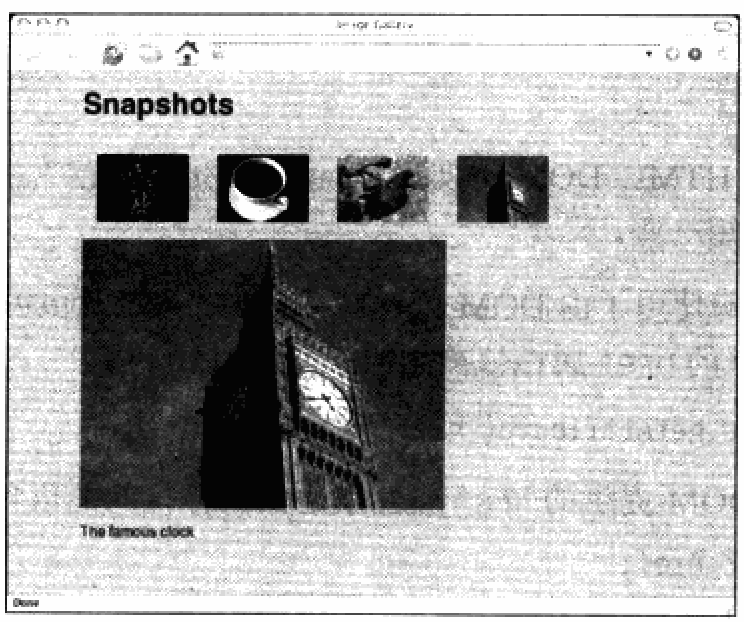
在本书的绝大多数章节里，我将只使用 DOM Core 来编写代码。虽然代码会因此而变得有点儿冗长，但我认为 DOM Core 方法更容易使用。当然，你们用不着也像我这样画地为牢，完全可以根据你们的个人喜好和具体情况来做出选择。我会尽可能地告诉你们，在哪些地方可以用 HTML-DOM 来简化代码。

6.6　小结

在这一章里，我对“JavaScript 美术馆”进行了多项优化。那些优化使我的 HTML 文档变得更加简明。我还为我的“JavaScript 美术馆”网页提供了一个基本的 CSS。当然，最重要的是我对我的 JavaScript 代码进行了改进。下面是我在这一章完成的几项主要工作：

- 为了让我的 JavaScript 代码不再依赖于那些没有保证的假设，我引入了许多项测试和检查。这些测试和检查使我的 JavaScript 代码有了足够的预留退路。
- 没有使用 onkeypress 事件处理函数，这使我的 JavaScript 代码的可访问性得到了保证。
- 最重要的是把事件处理函数从 HTML 文档分离到了一个外部的 JavaScript 文件。这使我的 JavaScript 代码不再依赖于 HTML 文档的内容和结构。

下面是我的“JavaScript 美术馆”在经过我在这一章里对它做出的种种优化之后的样子。



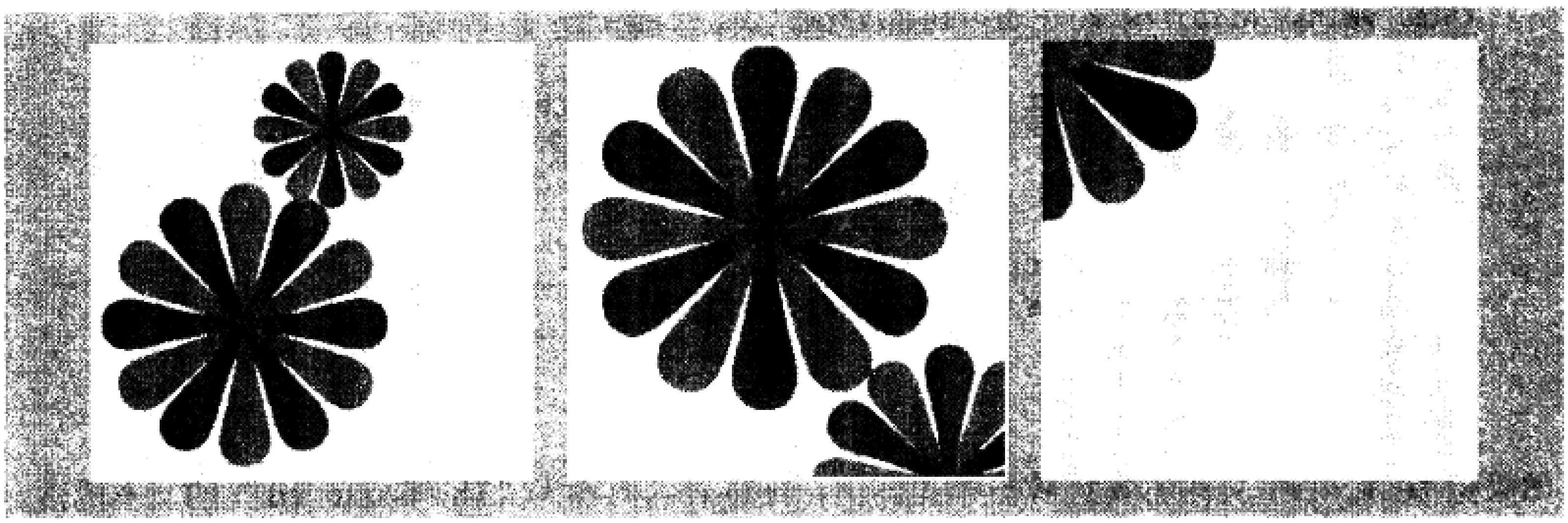
我认为：结构与行为的相互分离程度越大越好。

在“JavaScript 美术馆”的 HTML 文档部分还有一些内容让我感到不太满意。比如说，在我的 HTML 文档里，`placeholder` 和 `description` 元素是单纯为了 `showPic()` 函数而存在的，但不支持或禁用了 JavaScript 功能的浏览器也会把它们呈现出来，这就难免会给访问者带来不便甚至是困扰。

理想的情况是，让这两个元素只在遇到那些支持 DOM 的浏览器时才出现在 HTML 文档里。在下一章里，你们将会看到我是如何利用 DOM 提供的方法和属性去创建 HTML 元素，并把它们插入 HTML 文档的。

第7章

动态创建 HTML 内容



本章内容

- 用来动态创建 HTML 内容的“老”技巧：document.write()方法和 innerHTML 属性
- 深入剖析 DOM 方法：createElement()、createTextNode()、appendChild()和 insertBefore()。

你们此前见过的 DOM 方法都是用于对已经存在的 HTML 元素进行处理的。在这一章里，你们将会看到一些通过创建新元素和修改现有元素而改变网页结构的 DOM 方法。

此前见过的绝大多数 DOM 方法只能用来查找元素。getElementById()和 getElementsByTagName()方法都可以方便快捷地找到文档中的某个特定的元素节点，这些元素随后可以用诸如 setAttribute()（改变某个属性的值）和 nodeValue（改变某个元素节点所包含的文本）之类的方法和属性来处理。

我的“JavaScript 美术馆”就是用这些方法和属性实现的。showPic()函数先找出 id 属性值是 placeholder 和 description 的两个元素，然后刷新它们的内容。placeholder 元素的 src 属性是用 setAttribute()方法修改的，description 元素所包含的文本是用 nodeValue 属性修改的。在这两种情况里，都是对已经存在的元素做出修改的。

这是绝大多数 JavaScript 函数的工作原理。网页的结构由 HTML 文档负责创建，JavaScript 函数只用来改变 HTML 文档的某些细节而不改变其底层结构。

JavaScript 也可以用来改变网页的结构和内容。

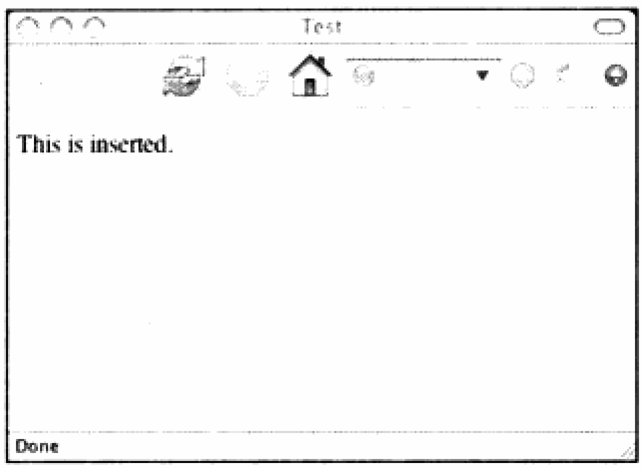
7.1 document.write()方法

document 对象的 write()方法可以方便快捷地把字符串插入到文档里。

请把以下 HTML 代码保存为一个文件，文件名就用 test.html 好了。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Test</title>
</head>
<body>
  <script type="text/javascript">
    document.write("<p>This is inserted.</p>");
  </script>
</body>
</html>
```

如果把 test.html 文件加载到 Web 浏览器里，你将看到内容为“This is inserted.”的文本段落。



document.write()方法的最大缺点是它违背了“分离 JavaScript”原则。即使把 document.write 语句挪到外部函数里，你也还是需要在 HTML 文档的<body>部分使用<script>标签才能调用那个函数。

下面这个函数以一个字符串为参数，它将把一个<p>标签、字符串和一个</p>标签拼接在一起。拼接后的字符串被保存到变量 str，然后用 document.write()方法写出来：

```
function insertParagraph(text) {
  var str = "<p>";
  str += text;
  str += "</p>";
  document.write(str);
}
```

你们可以把这个函数保存在外部文件 example.js 里。为了调用这个函数，你必须在 HTML 文档里插入<script>标签：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Test</title>
  <script type="text/javascript" src="example.js">
  </script>
</head>
<body>
  <script type="text/javascript">
    insertParagraph("This is inserted.");
  </script>
</body>
</html>
```

像上面这样把 JavaScript 和 HTML 代码混杂在一起是一种很不好的做法。这样的 HTML 文档既不容易阅读和编辑，也无法享受到把行为与结构分离开来的好处。

这样的文档还很容易导致“数据/格式非法”错误。比如说，在第一个例子里，<script>标签后面的“<p>”很容易被误认为是<p>标签，而在<script>标签的后面打开<p>标签是非法的。事实上，那个“<p>”和“</p>”只不过是一个将被插入 HTML 文档的字符串的组成部分而已。

还有，MIME 类型 application/xhtml+xml 与 document.write()方法不兼容，浏览器在呈现这种 XHTML 文档时根本不会执行 document.write()方法。

从某种意义上讲，使用 document.write()方法有点儿像使用<font>标签去设定字体和颜色。虽然这两种技巧在 HTML 文档里的使用效果都不错，但它们都算不上是最佳解决方案。

把结构、行为和样式分开永远都是一个好主意。只要有可能，就应该用外部 CSS 文件代替<font>标签去设定和管理网页的样式信息，就应该用外部 JavaScript 文件去控制网页的行为。你们应该避免在 HTML 文档的<body>部分使用<script>标签，避免使用 document.write()方法。

7.2 innerHTML 属性

现如今的浏览器几乎都支持属性 innerHTML，但这个属性并不是 W3C DOM 标准的组成部分。它始见于微软公司的 IE 4 浏览器，并从那时起被其他的浏览器接受。不过，从目前来看，它不太可能成为标准化 DOM 的组成部分。

innerHTML 属性可以用来读、写某给定元素里的 HTML 内容。请把下面这段 HTML 代码插入 test.html 文档的<body>部分：

```
<div id="testdiv">
<p>This is <em>my</em> content.</p>
</div>
```

div 元素的 id 属性值是 testdiv。它包含着一个元素节点（p 元素）。这个 p 元素又有一些子节点。有两个文本节点。这些文本节点的值是“This is”和“content”。还有一个元素节点（em 元素），em 元素本身包含着一个文本节点，这个文本节点的值是“my”。

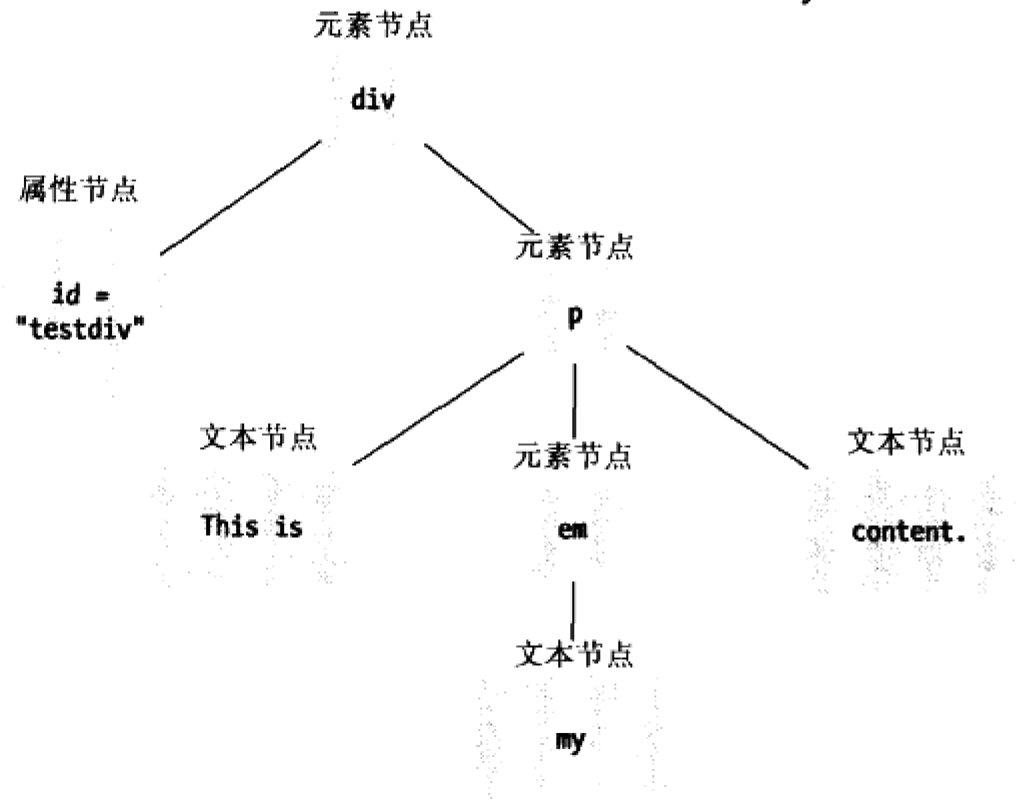


图 7-1　div 元素的 DOM 树。

图 7-1 中的 DOM 树可以让我们看到 div 元素的每个细节，而 DOM 提供的方法和属性可以让我们对这些节点当中的任何一个进行访问。

innerHTML 属性要简单得多，图 7-2 是 div 元素的 innerHTML 属性示意图：只有一个取值为 <p>This is my content.</p>的 HTML 字符串。

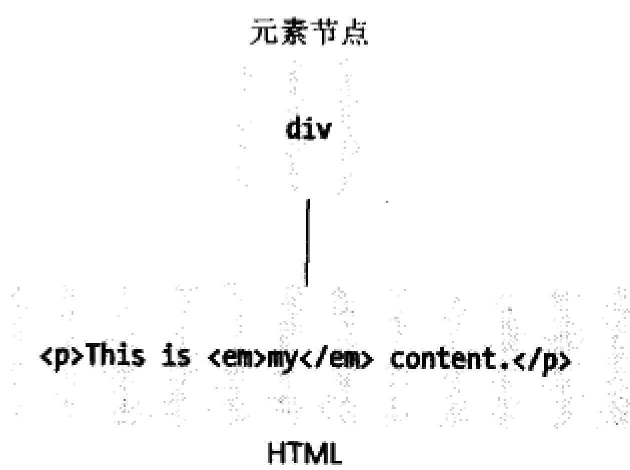
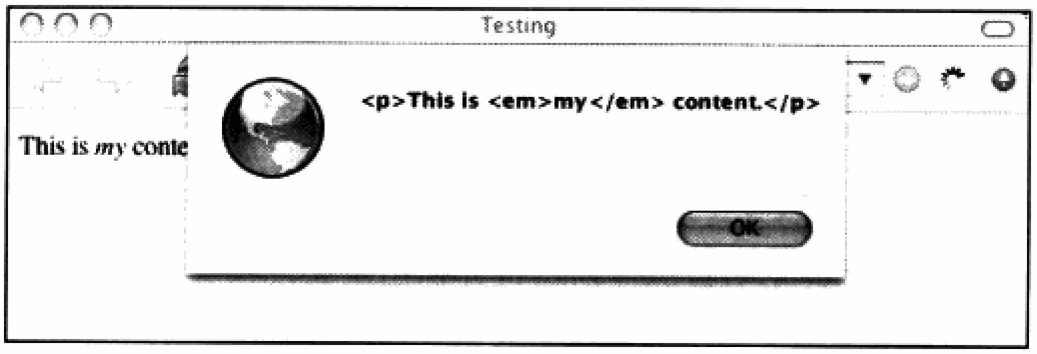


图 7-2　div 元素的 innerHTML 属性

可以通过先把下面这个新函数添加到 example.js 文件里：来检查一下这个 div 元素的 innerHTML 属性。

```
window.onload = function() {
  var testdiv = document.getElementById("testdiv");
  alert(testdiv.innerHTML);
}
```

然后在 Web 浏览器里刷新 test.html 页面，div 元素(它的 id 属性值等于 testdiv)的 innerHTML 属性值将显示在一个 alter 对话框里。



很明显，innerHTML 属性无细节可言。要想获得细节，就必须使用 DOM 方法和属性。标准化的 DOM 就像是一把手术刀，innerHTML 属性就像是一把劈柴斧。

劈柴斧有劈柴斧的用处。在你需要把一大段 HTML 内容插入一个网页时，innerHTML 属性更适用。它是一个读/写方法，你不仅可以用它来读出元素的 HTML 内容，还可以用它把 HTML 内容写入元素。

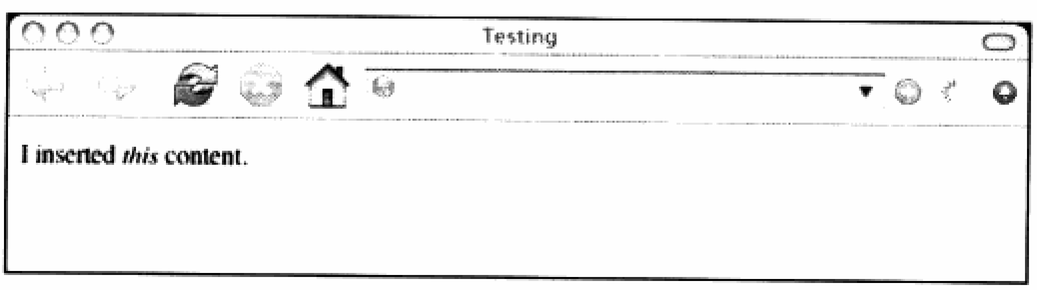
编辑 test.html 文件，让 id 属性值等于 testdiv 的元素变成空白：

```
<div id="testdiv">
</div>
```

如果把下面这段 JavaScript 代码放入 example.js 文件，就可以把一段 HTML 内容插入这个 <div>标签：

```
window.onload = function() {
  var testdiv = document.getElementById("testdiv");
  testdiv.innerHTML = "<p>I inserted <em>this</em> content.</p>";
}
```

在 Web 浏览器里刷新 test.html 文件，你就可以看到结果。



利用这个技巧无法区分“插入一段 HTML 内容”和“替换一段 HTML 内容”。testdiv 元素里有没有 HTML 内容无关紧要：一旦你使用了 innerHTML 属性，它的全部内容都将被替换。

在 test.html 文件里，把 id 属性值等于 testdiv 的元素的内容修改回它原来的样子：

```
<div id="testdiv">
<p>This is <em>my</em> content.</p>
</div>
```

example.js 文件保持不变。如果你在 Web 浏览器里刷新 test.html 文件，结果将和刚才一样。包含在 testdiv 元素里 HTML 内容被 innerHTML 属性完全改变了，原来的 HTML 内容未留下任何痕迹。

优点和缺点

在需要把一大段 HTML 内容插入一份文档时，innerHTML 属性可以让你又快又简单地完成这一任务。不过，innerHTML 属性不会返回任何可以让你用来对刚插入的内容进行处理的内容。如果你想对刚插入的内容进行处理，你将需要 DOM 方法提供的精确性。

innerHTML 属性要比 document.write()方法更值得推荐。使用了 innerHTML 属性，你就可以把 JavaScript 代码从 HTML 文档分离出来。用不着再在 HTML 文档的<body>部分插入<script>标签。

类似于 document.write()方法，innerHTML 属性也是只适用于 HTML 文档。浏览器在呈现正宗的 XHTML 文档（即 MIME 类型是 application/xhtml+xml 的 XHTML 文档）时不会去执行 innerHTML 属性。

还有一件事要提醒大家：innerHTML 属性是一项专利技术，不是一项业界标准。我认为，在编写 JavaScript 代码时应该避免使用任何形式的专利，这样我们才不会再次陷入又一场浏览器大战。在上一场这样的战争里（参请见第一章），不同品牌的浏览器使用的是不同的 DOM，这给程序员和用户带来了许多烦恼和不便。虽说 innerHTML 属性现在已经得到了广泛的支持，但它的未来仍是一个未知数。

在任何时候，标准的 DOM 都是可以替代 innerHTML 的。虽说这往往需要多编写一些代码才能获得同样的效果，但 DOM 提供了更高的精确性和更多的功能。

7.3　DOM 提供的方法

getElementById()和 getElementsByTagName()等方法可以把关于文档结构和内容的信息检索出来，它们非常有用。

DOM 把文档表示为一棵节点树。DOM 节点树所包含的信息与文档里的信息一一对应。你只要学会问正确的问题（使用正确的方法），就可以把 DOM 节点树上任何一个节点的细节检索出来。

DOM 是一条双向车道。不仅可以查询文档的内容，还可以刷新文档的内容。只要改变了 DOM 节点树，文档在浏览器里的呈现效果就会发生变化。

我们已经见识过 setAttribute()方法的神奇之处了：当用这个方法改变了 DOM 节点树上的某个属性节点时，有关文档在浏览器里呈现效果就会发生相应的变化。

不过，setAttribute()方法并未改变文档的物理内容：如果用文本编辑器而不是浏览器去打开有关文档，我们将看不到任何属性发生了变化。我们只有在用浏览器打开那份文档时才能看到文档呈现效果方面的变化。这是因为浏览器实际显示的是那棵 DOM 节点树。在浏览器看来，DOM 节点树才是文档。

一旦明白了这个道理，以动态方式实时创建 HTML 内容的事就变得不那么难以理解了——我们并不是在创建 HTML 内容，而是在改变 DOM 节点树。理解这一思路的关键是从 DOM 的角度去思考问题。

根据 DOM，一个文档就是一棵节点树。如果你想在节点树上添加内容，就必须插入新的节点。

如果你想把一些 HTML 内容添加到文档里，就必须在相应的 DOM 节点树上插入元素节点。

7.3.1 createElement()方法

首先，请编辑 test.html 文件，让 id 属性值等于 testdiv 的那个<div>标签的内容变成空白：

```
<div id="testdiv">
</div>
```

我想把一段文本插入 testdiv 元素。用 DOM 的术语来说，就是我想把一个 p 元素节点作为 div 元素节点的子节点添加到 DOM 节点树上（div 元素节点已经有了一个子节点：那是一个 id 属性节点，它的值是 testdiv）。

这项任务需要分两个步骤来进行：

(1) 创建一个新的元素。
(2) 把这个新元素插入节点树。

第一个步骤要用 DOM 方法 createElement 来完成。下面是这个方法的语法：

document.createElement(nodeName)

下面这条语句将创建一个 p 元素：

```
document.createElement("p");
```

这个方法本身并无实际用处——我们还需要把这个新创建出来的元素插入文档才能达到我们的最终目的。为此，我们需要有个东西来引用这个新创建出来的节点。作为一个编程套路，只要使用了 createElement()方法，就应该把新创建出来的元素赋值给一个变量：

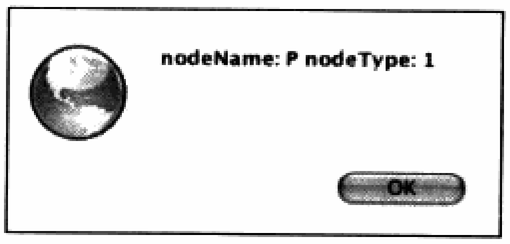
```
var para = document.createElement("p");
```

变量 para 现在包含着一个指向我们刚创建出来的那个 p 元素的引用指针。

现在，虽然这个新创建出来的 p 元素已经存在了，但它还不是任何一棵 DOM 节点树的组成部分，它只是游荡在 JavaScript 世界里的一个孤儿。它现在只是一个 DocumentFragment 对象，还无法显示浏览器的窗口画面里。不过，它已经像任何其他的节点那样有了自己的 DOM 属性。

这个无家可归的 p 元素现在已经有一个 nodeType 和一个 nodeName 值。这一事实可以用下面这段代码来验证（请把以下代码放入 example.js 文件并在浏览器里刷新 test.html 文档）：

```
window.onload = function() {
  var para = document.createElement("p");
  var info = "nodeName: ";
  info+= para.nodeName;
  info+= " nodeType: ";
  info+= para.nodeType;
  alert(info);
}
```



新节点确实已经存在，它有一个取值为 P 的 nodeName 属性。它还有一个取值为 1 的 nodeType 属性，而这意味着它是一个元素节点。不过，这个节点现在还未被挂接到 test.html 文档的节点树上。

7.3.2 appendChild()方法

把新创建的节点插入某个文档的节点树的最简单的办法是，让它成为这个文档的某个现有节点的一个子节点。

具体到这个例子，我想把一段文本插入到 test.html 文档中的那个 id 属性值是 testdiv 的元素节点。换句话说，我想让新创建的 p 元素成为 testdiv 元素的一个子节点。可以用 DOM 提供的 appendChild()方法来完成这一任务。

下面是 appendChild()方法的语法：

parent.appendChild(child)

具体到 test.html 文档这个例子，上面这个语法中的 *child* 就是我刚才用 createElement()方法创建出来的，*parent* 就是那个 id 属性值是 testdiv 的元素节点。我将需要用一个 DOM 方法把“testdiv”节点提取出来；提取这个节点最简单的办法是使用 getElementById()方法。

像往常一样，我将把这个元素赋值给一个变量，这可以让我的代码简明易读：

```
var testdiv = document.getElementById("testdiv");
```

变量 para 现在包含着一个指向那个 id 属性值等于 testdiv 的元素的引用指针。

在上一小节我创建了一个 para 变量，它包含着一个指向我们刚创建的那个 p 元素的引用指针：

```
var para = document.createElement("p");
```

有了这些，我就可以像下面这样用 appendChild()方法把变量 para 插入变量 testdiv 了：

```
testdiv.appendChild(para);
```

新创建的 p 元素现在成为了 testdiv 元素的一个子节点。它不再是 JavaScript 世界里的一个孤儿，它已经被插入到 test.html 文档的节点树里了。

在使用 appendChild()方法时，我们不必非得使用一些变量来引用父节点和子节点。事实上，完全可以把上面这条语句写成下面这样：

```
document.getElementById("testdiv").appendChild(
➥ document.createElement("p"));
```

可以看到，上面这样的代码很难阅读和理解。下面这些代码虽然需要多打一些字，但从长远来讲是值得的：

```
var para = document.createElement("p");
var testdiv = document.getElementById("testdiv");
testdiv.appendChild(para);
```

7.3.3 createTextNode()方法

我现在已经创建出了一个元素节点并把它插入了文档的节点树。

我创建出来的这个节点是一个空白的 p 元素。我想把一些文本放入这个 p 元素，但 createElement()方法帮不了我这个忙——它只能创建元素节点。我需要创建一个文本节点，我可以用 createTextNode 方法来实现它。

注意 千万不要把这些方法的名字混淆了。它们的名字是createElement和createTextNode——千万不要把前者误写为createElementNode、把后者误写为createText！

---

createTextNode()方法的语法与 createElement()方法的很相似：

document.createTextNode(text)

下面这条语句将创建出一个内容为“Hello world”的文本节点：

```
document.createTextNode("Hello world");
```

和刚才一样，我决定把这个新创建的节点也赋值给一个变量：

```
var txt = document.createTextNode("Hello world");
```

变量 txt 现在包含着一个指向新创建的那个文本节点的引用指针。这个节点现在还是 JavaScript 世界里的一个孤儿，因为它还未被插入任何一个文档的节点树。

我可以用 appendChild()方法把这个文本节点插入为某个现有元素的子节点。我将把这个文本节点插入到我在上一小节创建的 p 元素。因为在上一小节里我已经把那个 p 元素存入了变量 para、现在又把新创建的文本节点存入了变量 txt，所以现在可以用下面这条语句来达到我的目的：

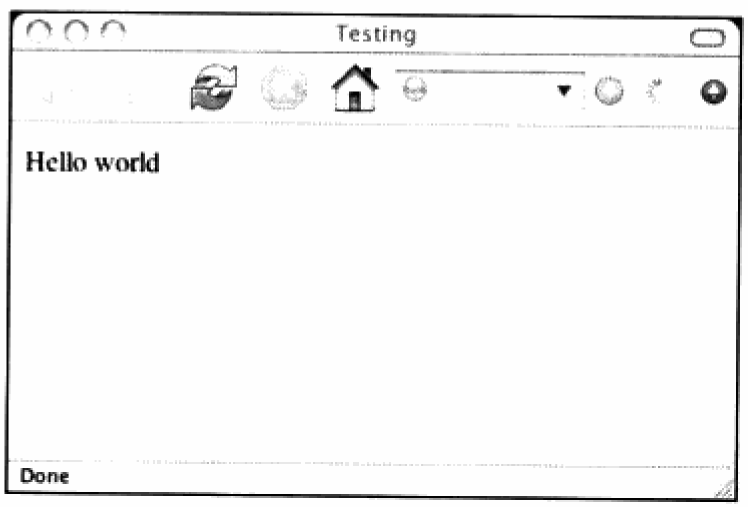
```
para.appendChild(txt);
```

内容为“Hello world”的文本节点就成为那个 p 元素的一个子节点了。

现在，试着把下面这段代码写入 example.js 文件：

```
window.onload = function() {
  var para = document.createElement("p");
  var testdiv = document.getElementById("testdiv");
  testdiv.appendChild(para);
  var txt = document.createTextNode("Hello world");
  para.appendChild(txt);
}
```

然后在浏览器里重新加载 test.html 文件，你们就会从浏览器窗口里看到文本“Hello world”，如下所示。



在这个例子里，我是按照以下顺序来创建和插入新节点的：

(1) 创建一个 p 元素节点。
(2) 把这个 p 元素节点追加到 test.html 文档中的一个元素节点上。
(3) 创建一个文本节点。
(4) 把这个文本节点追加到刚才创建的那个 p 元素节点上。

appendChild()方法还可以用来连接那些尚未成为文档树的组成部分的节点。也就是说，以下步骤同样可以让我达到目的：

(1) 创建一个 p 元素节点。

(2) 创建一个文本节点。

(3) 把这个文本节点追加到第 1 步创建的 p 元素节点上。

(4) 把这个 p 元素节点追加到 test.html 文档中的一个元素节点上。

下面是按照新步骤编写出来的函数：

```
window.onload = function() {
  var para = document.createElement("p");
  var txt = document.createTextNode("Hello world");
  para.appendChild(txt);
  var testdiv = document.getElementById("testdiv");
  testdiv.appendChild(para);
}
```

最终的结果是一样的。把上面这些代码写入 example.js 文件，并在浏览器里重新加载 test.html 文件。你们将看到文本“Hello world”——就像刚才一样。

一个更复杂的组合

刚才介绍 innerHTML 属性时，我使用了如下所示的 HTML 内容：

```
<p>This is <em>my</em> content.</p>
```

与创建一个包含着一些文本的 p 元素相比，把上面这样的 HTML 内容插入文档的步骤要复杂不少。为了把这些 HTML 内容插入 test.html 文档，我决定先把它转换为一棵节点树。

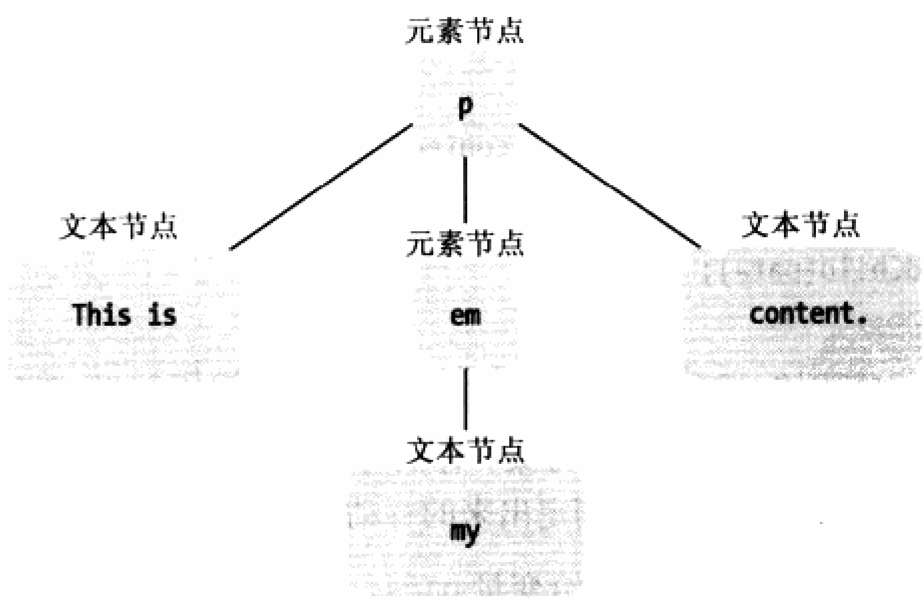


图 7-3 把一段 HTML 内容转换为一棵节点树

如图 7-3 所示，这些 HTML 内容对应着一个 p 元素节点，它本身又包含着以下子节点：

- ❑ 一个文本节点，其内容是“This is”
- ❑ 一个元素节点，这个元素节点的名字是“em”；这个元素节点本身还包含着：
  - ■ 一个文本节点，其内容是“my”

- 一个文本节点，其内容是“ content.”（第一个字符是空格，最后一个字符是英文句号）。

把 HTML 内容转换为节点树的目的是，把需要创建哪些节点的问题弄清楚。只有把这个问题弄清楚，我们才能制定出一个妥善的行动计划：

(1) 创建一个 p 元素节点并把它赋值给变量 para。

(2) 创建一个文本节点并把它赋值给变量 txt1。

(3) 把 txt1 追加到 para 上。

(4) 创建一个 em 元素节点并把它赋值给变量 emphasis。

(5) 创建一个文本节点并把它赋值给变量 txt2。

(6) 把 txt2 追加到 emphasis 上。

(7) 把 emphasis 追加到 para 上。

(8) 创建一个文本节点并把它赋值给变量 txt3。

(9) 把 txt3 追加到 para 上。

(10) 把 para 追加到 test.html 文档中的 testdiv 元素上。

下面是根据上述步骤编写出来的 JavaScript 代码：

```
window.onload = function() {
  var para = document.createElement("p");
  var txt1 = document.createTextNode("This is ");
  para.appendChild(txt1);
  var emphasis = document.createElement("em");
  var txt2 = document.createTextNode("my");
  emphasis.appendChild(txt2);
  para.appendChild(emphasis);
  var txt3 = document.createTextNode(" content.");
  para.appendChild(txt3);
  var testdiv = document.getElementById("testdiv");
  testdiv.appendChild(para);
}
```

把上面这些代码写入 example.js 文件，然后在浏览器里重新加载 test.html 文档。

如果你们愿意，可以采用一种不同的方案。你们可以先把所有的节点都创建出来，然后再把它们连接在一起。下面是按照这一思路制定出来的行动计划：

(1) 创建一个 p 元素节点并把它赋值给变量 para。

(2) 创建一个文本节点并把它赋值给变量 txt1。

(3) 创建一个 em 元素节点并把它赋值给变量 emphasis。

(4) 创建一个文本节点并把它赋值给变量 txt2。

(5) 创建一个文本节点并把它赋值给变量 txt3。

(6) 把 txt1 追加到 para 上。

(7) 把 txt2 追加到 emphasis 上。

(8) 把 emphasis 追加到 para 上。
(9) 把 txt3 追加到 para 上。
(10) 把 para 追加到 test.html 文档中的 testdiv 元素上。

下面是根据上述步骤编写出来的 JavaScript 代码：

```
window.onload = function() {
  var para = document.createElement("p");
  var txt1 = document.createTextNode("This is ");
  var emphasis = document.createElement("em");
  var txt2 = document.createTextNode("my");
  var txt3 = document.createTextNode(" content.");
  para.appendChild(txt1);
  emphasis.appendChild(txt2);
  para.appendChild(emphasis);
  para.appendChild(txt3);
  var testdiv = document.getElementById("testdiv");
  testdiv.appendChild(para);
}
```

如果把上面这些代码写入 example.js 文件，然后在浏览器里重新加载 test.html 文档，你们将看到与前面一模一样的结果。

可以看到，把新节点插入某个文档的节点树的办法并非只有一种。即使你们决定永远也不使用 document.write()方法或 innerHTML 属性，在使用 DOM 方法去创建和插入新节点时你们也可以灵活地做出多种选择。

7.4 重回“JavaScript 美术馆”

现在，请把 test.html 和 example.js 文件放在一边。我将向你们展示一个动态创建 HTML 内容的实用案例。

在上一章里，我对“JavaScript 美术馆”脚本做了许多改进。我把 JavaScript 代码分离出了 HTML 文档，并为 JavaScript 代码预留了退路，我还做了一些与网页的可访问性有关的改进。

不过，在“JavaScript 美术馆”的 HTML 文档部分还有一些内容让我感到不太满意。我的 gallery.html 文件目前包含着以下 HTML 内容：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Image Gallery</title>
  <script type="text/javascript" src="scripts/showPic.js"></script>
  <link rel="stylesheet" href="styles/layout.css" type="text/css"
➥ media="screen" />
```

```
</head>
<body>
  <h1>Snapshots</h1>
  <ul id="imagegallery">
    <li>
      <a href="images/fireworks.jpg" title="A fireworks display">
        <img src="images/thumbnail_fireworks.jpg" alt="Fireworks" />
      </a>
    </li>
    <li>
      <a href="images/coffee.jpg" title="A cup of black coffee" >
        <img src="images/thumbnail_coffee.jpg" alt="Coffee" />
      </a>
    </li>
    <li>
      <a href="images/rose.jpg" title="A red, red rose">
        <img src="images/thumbnail_rose.jpg" alt="Rose" />
      </a>
    </li>
    <li>
      <a href="images/bigben.jpg" title="The famous clock">
        <img src="images/thumbnail_bigben.jpg" alt="Big Ben" />
      </a>
    </li>
  </ul>
  <img id="placeholder" src="images/placeholder.gif"
➥ alt="my image gallery" />
  <p id="description">Choose an image.</p>
</body>
</html>
```

这个 XHTML 文件中的 placeholder 和 description 元素是单纯为了 showPic 脚本而存在的。我想把结构和行为彻底分开。既然这些元素的存在意义只是为了让一些 DOM 方法能够对它们进行处理，那么用 DOM 方法来创建它们才是最合理的选择。

第一步非常简单：把这些元素从 gallery.html 文档里删掉。此后，我将编写一些 JavaScript 代码把它们动态地创建出来。

我将编写一个函数 preparePlaceholder 并把它放到 showPic.js 文件里。我将在文档加载时调用这个函数。下面是这个函数将要完成的任务：

(1) 创建一个 img 元素节点。
(2) 设置这个节点的 id 属性。
(3) 设置这个节点的 src 属性。
(4) 设置这个节点的 alt 属性。
(5) 创建一个 p 元素节点。

(6) 设置这个节点的 id 属性。
(7) 创建一个文本节点。
(8) 把这个文本节点追加到 p 元素上。
(9) 把 p 元素和 img 元素插入到 gallery.html 文档。

创建这些元素和设置各有关属性的工作不需要再做解释了。我在这里组合使用了 createElement()、createTextNode()和 setAttribute()方法：

```
var placeholder = document.createElement("img");
placeholder.setAttribute("id","placeholder");
placeholder.setAttribute("src","images/placeholder.gif");
placeholder.setAttribute("alt","my image gallery");
var description = document.createElement("p");
description.setAttribute("id","description");
var desctext = document.createTextNode("Choose an image");
```

接下来，我将用 appendChild()方法把新创建的文本节点插入 p 元素：

```
description.appendChild(desctext);
```

最后一步是把新创建的元素插入文档。很凑巧，因为“JavaScript 美术馆”的图片清单（ ...)刚好是文档中的最后一个元素，所以如果把 placeholder 和 description 元素追加到 body 元素节点上，它们就会出现在图片清单的后面。我可以通过标签名“body”把 body 标签当作第一个（唯一的）元素引用。

```
document.getElementsByTagName("body")[0].appendChild(placeholder);
document.getElementsByTagName("body")[0].appendChild(description);
```

当然，也可以使用 HTML-DOM 提供的快捷方式——body 属性，这将把上面两条语句简化为：

```
document.body.appendChild(placeholder);
document.body.appendChild(description);
```

这两组语句都将把 placeholder 和 description 元素插入到位于文档末尾的</body>标签的前面。

以上代码实现了我的想法，但这一切都依赖于一个细节：“JavaScript 美术馆”的图片清单刚好是<body>部分的最后一个元素。如果在这个图片清单的后面还有一些其他的内容该怎么办？我的真实想法是，让新创建的元素紧跟在图片清单的后面，不管这个清单本身出现在文档中的什么地方。

7.4.1 insertBefore()方法

DOM 提供了名为 insertBefore()方法，这个方法将把一个新元素插入到一个现有元素的前面。在调用此方法时，我们必须告诉它三件事：

(1) 想插入的新元素（*newElement*）。
(2) 想把这个新元素插入到哪个现有元素（*targetElement*）的前面。

(3) 这两个元素共同的父元素（*parentElement*）。

下面是这个方法的调用语法：

parentElement.insertBefore(newElement,targetElement)

我们不必知道那个父元素到底是哪个，因为 *targetElement* 元素的 parentNode 属性值就是它。在 DOM 里，元素节点的父元素必须是另一个元素节点（属性节点和文本节点的子元素不允许是元素节点）。

比如说，我可以用下面这条语句把 placeholder 和 description 元素插入到图片清单的前面（还记得吗，图片清单的 id 属性值是 imagegallery）：

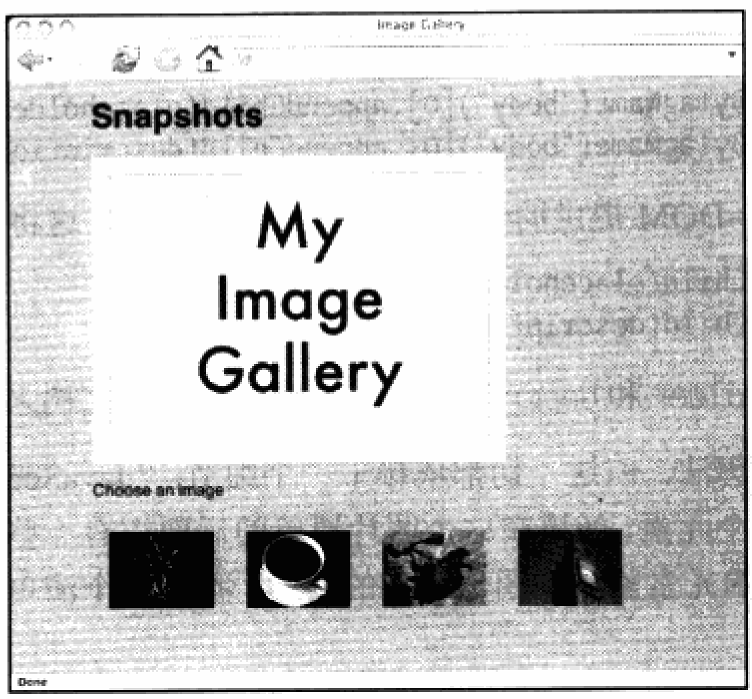
```
var gallery = document.getElementById("imagegallery");
gallery.parentNode.insertBefore(placeholder,gallery);
```

此时，变量 gallery 的 parentNode 属性值是 body 元素，所以 placeholder 元素将被插入为 body 元素的新子元素，它将被插入到它的兄弟元素 gallery 的前面。

还可以把 description 元素插入为 gallery 元素的一个兄弟元素：

```
gallery.parentNode.insertBefore(description,gallery);
```

在 gallery 清单的前面插入 placeholder 图片和 description 文本段的效果如下图所示。



这种效果其实也不错，但我刚才说的是把新创建的元素插入到图片清单的后面，不是前面。

你们很可能会这么想：既然有一个 insertBefore()方法，就应该有一个相应的 insertAfter()方法。很可惜，这种想法是错的——DOM 根本没有提供这样的方法。

1. 编写insertAfter()函数

我不知道 DOM 为何未提供一个 insertAfter()方法。我认为这样的方法会非常有用。

虽然 DOM 本身没有提供 insertAfter()方法，但我们完全可以利用现有的 DOM 方法和属性把一个节点插入到另一个节点的后面。下面是我编写出来的 insertAfter()函数：

```
function insertAfter(newElement,targetElement) {
  var parent = targetElement.parentNode;
  if (parent.lastChild == targetElement) {
    parent.appendChild(newElement);
  } else {
    parent.insertBefore(newElement,targetElement.nextSibling);
  }
}
```

这个函数用到了以下 DOM 方法和属性：

- parentNode 属性
- lastChild 属性
- appendChild()方法
- insertBefore()方法
- nextSibling 属性

下面，请大家随我一起去看看这个函数是如何一步一步地完成它的工作的：

首先，这个函数有两个参数：一个是将被插入的新元素，另一个是新元素将被插入到它前面去的目标元素。这两个参数将通过变量 newElement 和 targetElement 被传递到这个函数里：

```
function insertAfter(newElement,targetElement)
```

把目标元素的 parentNode 属性值提取到变量 parent 里：

```
var parent = targetElement.parentNode
```

接下来，检查目标元素是不是 parent 的最后一个子元素，即比较 parent 元素的 lastChild 属性值与目标元素是否存在“等于”关系：

```
if (parent.lastChild == targetElement)
```

如果是，就用 appendChild()方法把新元素追加到 parent 元素上，这样就恰好把新元素紧跟着插入到目标元素的后面了：

```
parent.appendChild(newElement)
```

如果不是，就把新元素插入到目标元素和 parent 元素的下一个子元素的中间。目标元素后面的下一个兄弟节点是目标元素的 nextSibling 属性。用 insertBefore()方法把新元素插入到目标元素的下一个兄弟节点的前面：

```
parent.insertBefore(newElement,targetElement.nextSibling)
```

这是一个相对比较复杂的小函数，但如果把它分解开来的话，读懂它也不是很难。就算你现在还不是完全明白也不要紧。等你对 insertAfter()函数所用到的 DOM 方法和属性更加熟悉时，你

自然会把它弄明白。

类似于 addLoadEvent()函数，insertAfter()函数也非常实用，应该把它们都收录到你们的脚本里。

2. 使用insertAfter()函数

我将把 insertAfter()函数用在我的 preparePlaceholder()函数里。首先，把图片清单检索出来：

```
var gallery = document.getElementById("imagegallery");
```

接下来，把 placeholder（这个变量对应着新创建出来的 img 元素）插入到 gallery 的后面：

```
insertAfter(placeholder,gallery);
```

placeholder 图片现在成了 gallery.html 文档的节点树的组成部分，而我还希望把 description 文本段插入到它的后面。我已经把这个节点保存在变量 description 里了。再次使用 insertAfter()方法，但这次是把 description 插入到 placeholder 的后面：

```
insertAfter(description, placeholder);
```

增加了上面这几条语句之后，preparePlaceholder()函数变成了如下所示的样子：

```
function preparePlaceholder() {
  var placeholder = document.createElement("img");
  placeholder.setAttribute("id","placeholder");
  placeholder.setAttribute("src","images/placeholder.gif");
  placeholder.setAttribute("alt","my image gallery");
  var description = document.createElement("p");
  description.setAttribute("id","description");
  var desctext = document.createTextNode("Choose an image");
  description.appendChild(desctext);
  var gallery = document.getElementById("imagegallery");
  insertAfter(placeholder,gallery);
  insertAfter(description,placeholder);
}
```

事情还未结束，这个函数还有最后一个问题没有解决：我在新增加的那几条语句里使用了一些新的 DOM 方法，但没有对浏览器是否支持它们进行测试和检查。为了确保这个函数有足够的退路，还需要再增加几条语句：

```
function preparePlaceholder() {
  if (!document.createElement) return false;
  if (!document.createTextNode) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("imagegallery")) return false;
```

```
  var placeholder = document.createElement("img");
  placeholder.setAttribute("id","placeholder");
  placeholder.setAttribute("src","images/placeholder.gif");
  placeholder.setAttribute("alt","my image gallery");
  var description = document.createElement("p");
  description.setAttribute("id","description");
  var desctext = document.createTextNode("Choose an image");
  description.appendChild(desctext);
  var gallery = document.getElementById("imagegallery");
  insertAfter(placeholder,gallery);
  insertAfter(description,placeholder);
}
```

7.4.2 “JavaScript 美术馆”二次改进版

现在我的 showPic.js 文件包含着 5 个不同的函数，它们是：

- addLoadEvent()函数
- insertAfter()函数
- preparePlaceholder()函数
- prepareGallery()函数
- showPic()函数

addLoadEvent()和 insertAfter()属于通用型函数，它们在许多场合都能派上用场。

preparePlaceholder()函数负责创建一个 img 元素和一个 p 元素。这个函数将把这些新创建的元素插入到 gallery.html 文档的节点树里——当然，新元素会被插入到 imagegallery 清单的后面。

prepareGallery()函数负责处理有关的事件。这个函数将遍历 imagegallery 清单里的每个链接，并给它加上一个 onclick 事件处理函数。当用户点击这些链接中的某一个时，就会调用 showPic()函数。

当用户点击了图片清单里的某个链接时，showPic()函数将把“占位符”图片切换为用户想要查看的图片。

为了激活这些功能，我用 addLoadEvent()函数把 preparePlaceholder()和 prepareGallery()函数绑定在了 onload 事件处理函数上。

```
addLoadEvent(preparePlaceholder);
addLoadEvent(prepareGallery);
```

下面是最终完成的 showPic.js 文件的内容：

```
function addLoadEvent(func) {
  var oldonload = window.onload;
  if (typeof window.onload != 'function') {
    window.onload = func;
```

```
  } else {
    window.onload = function() {
      oldonload();
      func();
    }
  }
}

function insertAfter(newElement,targetElement) {
  var parent = targetElement.parentNode;
  if (parent.lastChild == targetElement) {
    parent.appendChild(newElement);
  } else {
    parent.insertBefore(newElement,targetElement.nextSibling);
  }
}

function preparePlaceholder() {
  if (!document.createElement) return false;
  if (!document.createTextNode) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("imagegallery")) return false;
  var placeholder = document.createElement("img");
  placeholder.setAttribute("id","placeholder");
  placeholder.setAttribute("src","images/placeholder.gif");
  placeholder.setAttribute("alt","my image gallery");
  var description = document.createElement("p");
  description.setAttribute("id","description");
  var desctext = document.createTextNode("Choose an image");
  description.appendChild(desctext);
  var gallery = document.getElementById("imagegallery");
  insertAfter(placeholder,gallery);
  insertAfter(description,placeholder);
}

function prepareGallery() {
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("imagegallery")) return false;
  var gallery = document.getElementById("imagegallery");
  var links = gallery.getElementsByTagName("a");
  for ( var i=0; i < links.length; i++) {
    links[i].onclick = function() {
      return showPic(this);
      }
    links[i].onkeypress = links[i].onclick;
  }
}
```

```
function showPic(whichpic) {
  if (!document.getElementById("placeholder")) return true;
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
  if (!document.getElementById("description")) return false;
  if (whichpic.getAttribute("title")) {
    var text = whichpic.getAttribute("title");
  } else {
    var text = "";
  }
  var description = document.getElementById("description");
  if (description.firstChild.nodeType == 3) {
    description.firstChild.nodeValue = text;
  }
  return false;
}

addLoadEvent(preparePlaceholder);
addLoadEvent(prepareGallery);
```

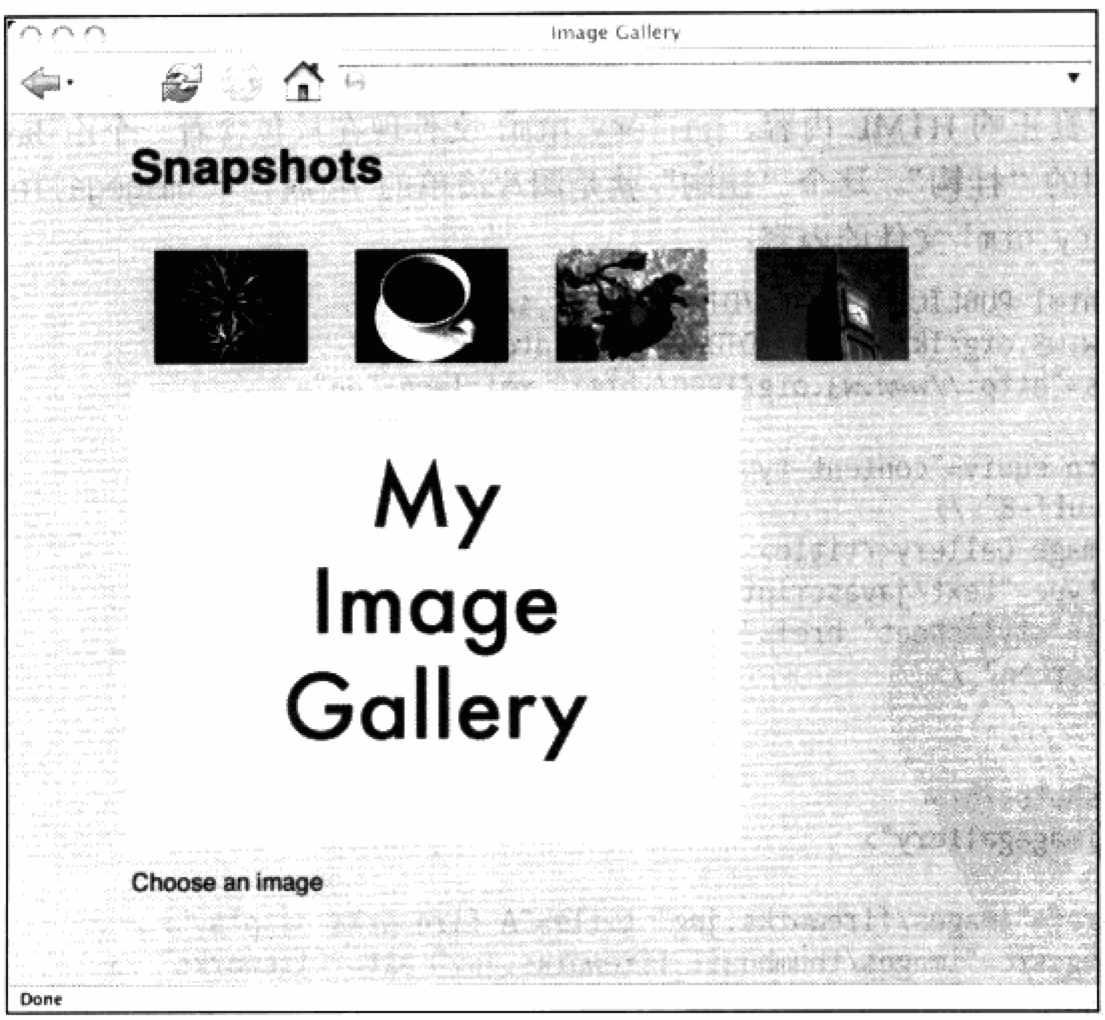
showPic.js 文件里的 JavaScript 代码增加了，但换来的是 gallery.html 文件里的 HTML 代码减少了。除了真正的 HTML 内容，gallery.html 文件现在只包含着一个由 JavaScript 脚本和 CSS 样式表共用的“挂钩”。这个“挂钩”就是图片清单的 id 属性（imagegallery）。下面是最终完成的 gallery.html 文件的内容：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;
➥ charset=utf-8" />
  <title>Image Gallery</title>
  <script type="text/javascript" src="scripts/showPic.js"></script>
  <link rel="stylesheet" href="styles/layout.css" type="text/css"
➥ media="screen" />
</head>
<body>
  <h1>Snapshots</h1>
  <ul id="imagegallery">
    <li>
      <a href="images/fireworks.jpg" title="A fireworks display">
        <img src="images/thumbnail_fireworks.jpg" alt="Fireworks" />
      </a>
    </li>
    <li>
      <a href="images/coffee.jpg" title="A cup of black coffee" >
        <img src="images/thumbnail_coffee.jpg" alt="Coffee" />
      </a>
```

```
      </li>
      <li>
        <a href="images/rose.jpg" title="A red, red rose">
          <img src="images/thumbnail_rose.jpg" alt="Rose" />
        </a>
      </li>
      <li>
        <a href="images/bigben.jpg" title="The famous clock">
          <img src="images/thumbnail_bigben.jpg" alt="Big Ben" />
        </a>
      </li>
    </ul>
  </body>
  </html>
```

现在，“JavaScript 美术馆”的结构、样式和行为已经不再纠缠不清了。

请把 `gallery.html` 文件加载到 Web 浏览器里。你们将看到 placeholder 图片和 description 文本段，它们是我用 JavaScript 代码插入到 imagegallery 清单后面的。



JavaScript 代码动态地创建出了这些 HTML 内容并把它们动态地添加到了文档里。

JavaScript 代码还对图片清单里的所有链接进行了预处理。你们可以点击那些缩微图中的任何一个去体验一下这个与众不同的“JavaScript 美术馆”。



7.5 小结

在这一章里，我们向大家介绍了几种不同的向浏览器里的文档动态添加 HTML 内容的办法。我们简要地回顾了两种“古老的”技巧：

- document.write()方法
- innerHTML 属性

我们详细深入地探讨了一些利用 DOM 方法来动态创建 HTML 内容的例子。运用这些方法的关键是，要学会把一份 Web 文档转换为一棵 DOM 节点树的要领。我们重点介绍了以下几种 DOM 方法：

- createElement()方法
- createTextNode()方法
- appendChild()方法
- insertBefore()方法

请记住，你用 createElement()或 createTextNode()方法刚刚创建出来的节点只是 JavaScript 世界里的孤儿，你还需要用 appendChild()或 insertBefore()方法把这些 DocumentFragment 对象插入某个文档的节点树才能让它们呈现在浏览器窗口里。

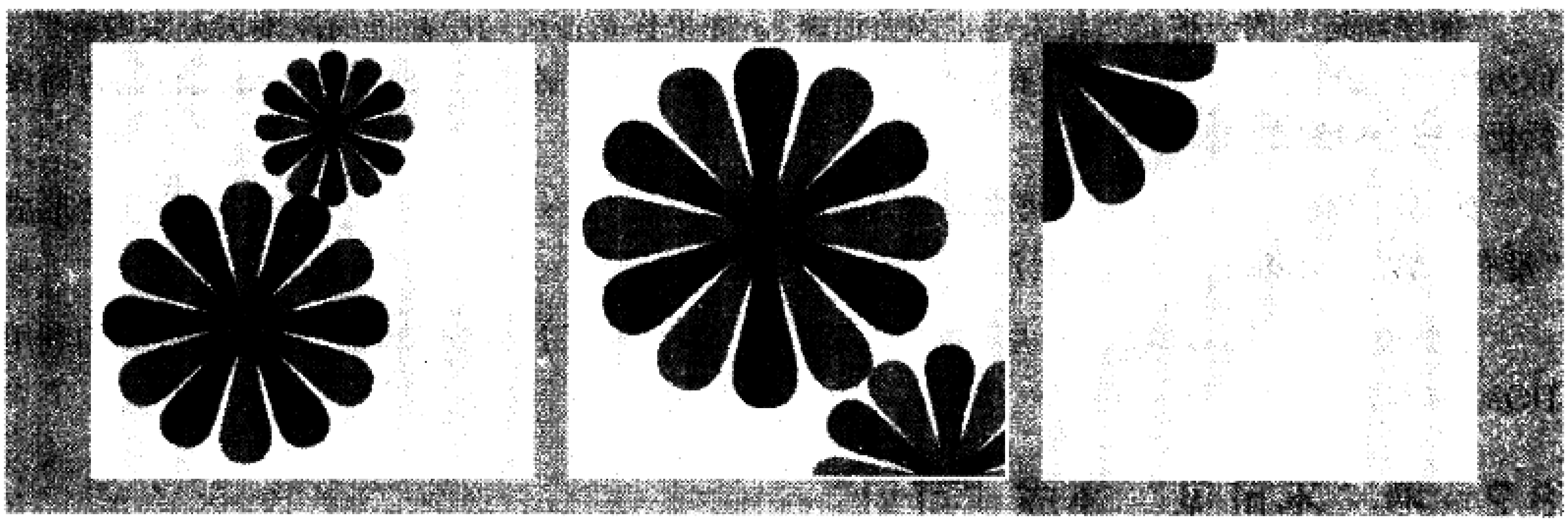
在这一章里，你们还看到了我是如何对“JavaScript 美术馆”做进一步改进的。在这一过程中，我还向大家介绍了一个非常实用的 `insertAfter()`函数。在需要把一些 HTML 内容动态地添加到一份文档里去的场合，这个函数往往能帮上你们的大忙。

7.6 下章简介

在下一章里，你们将会看到更多向文档动态添加 HTML 内容的例子。文档里的某些信息在浏览器窗口里通常是看不到的，但“看不到”并不意味着“没有用”。在下一章里，我将向大家展示怎样才能把那些“不可见”的信息动态地创建为 HTML 内容并把它们动态地添加到文档上去。

第 8 章

充实文档的内容



本章内容

- 一个为文档创建“缩略词语表”的函数
- 一个为文档创建“文献来源链接”的函数
- 一个为文档创建“快速访问键清单”的函数

在上一章里，我们向大家介绍了一些利用 DOM 方法和属性来动态创建 HTML 内容的技巧。在这一章里，我们将把那些技巧运用到实践中。我们将运用那些 DOM 方法和属性去创建一些有用的 HTML 内容并把它们添加到网页里。

8.1 不应该做的事情

我们可以用 JavaScript 把内容插入一份文档，但“可以做一件事”并不意味着“应该做这件事”。

从理论上讲，完全可以用 JavaScript 把一些重要的内容添加到网页上。但在实际工作中，这种做法非常不好，因为那会让你的 JavaScript 脚本根本没有空间去预留退路。那些缺乏必要的 JavaScript 支持的访问者将永远也看不到你心目中的重要内容。别忘了，至少在目前，各大搜索引擎网站的搜索机（searchbot）还不怎么支持 JavaScript。

如果你觉察到自己正在使用 DOM 技术把一些重要的内容添加到网页上，则应该立刻停下来去检讨你的计划和思路。你很可能会发现自己正在滥用 DOM 技术！

无论何时何地，都要把“循序渐进”和“预留退路”这两项原则牢记在心中。

所谓循序渐进 progressive enhancement 原则说的是这样一种思考问题的方法：从最核心的内容开始，逐步添加额外的功能。应该先用标记语言给核心内容加上正确的标记以使其获得正确的结构；然后再逐步充实被加上了正确标记的内容。充实的内容既可以是通过 CSS 样式表实现各种呈现效果，也可以是通过 DOM 脚本添加各种操作行为。

如果你正在使用 DOM 技术添加核心内容，那么你添加它们的时机未免太迟了——核心内容应该在刚开始编写文档时就成为文档的组成部分。

“循序渐进”的一种必然结果是“预留退路”。如果你是按照循序渐进的原则去充实内容的，你为内容添加的样式和行为就会有足够的预留退路，那些缺乏必要的 CSS 和 DOM 支持的访问者就仍能够访问到你的核心内容。

如果用 JavaScript 去添加重要的内容，你将很难保证它们有足够的预留退路。这往往会形成“没有足够的 JavaScript 支持，就没有内容可看”的尴尬局面。

如此说来，利用 DOM 技术去添加内容是不是一种禁忌呢？不是，有很多场合需要我们利用 DOM 技术去生成内容。

8.2 把“不可见”变成“可见”

现如今的 Web 设计人员能够从许多方面对网页的显示效果加以控制。在对包容在(X)HTML 标记内的内容进行排版方面，CSS 提供了非常强大的功能。这种技术早已超越了对网页内容的字体和颜色进行简单调整的初级阶段。利用 CSS，我们可以把原本纵向排列的元素显示成一行。“JavaScript 美术馆”页面上由缩略图构成的图片清单就是一个很好的例子。包容在<li>标签里的列表项在通常情况下各占一行，但在我把每个列表项的 display 属性设置为 inline 之后，那些列表项在浏览器窗口里从纵向排列变成了横向排列。

反过来也是可以的。对于通常是横向排列的元素，我们只须把它的 display 属性设置为 block，就可以让这个元素独占一行。如果把某个元素的 display 属性设置为 none，我们甚至可以让它根本不出现在浏览器窗口里——这个元素仍是 DOM 节点树的组成部分，只是浏览器不显示它们而已。

CSS 的功能确实强大，但它并不是万能的。CSS 只能对包容在(X)HTML 标记内的内容进行排版处理——这至少在目前这个时期是如此。CSS 目前只能根据各有关元素的 id 和 class 属性来改变它们在浏览器窗口里的呈现效果。

在把一些内容放到(X)HTML 标记之间时，我们其实是在使用那些标记对那些内容的结构和语意做出定义。在对内容进行标记时，把正确的属性设置为正确的值是这项工作的重要组成部分。

绝大多数属性的内容（即属性值）在 Web 浏览器里都是看不到的，只有极少数属性属于例

外，但不同的浏览器在显示这些例外的属性时往往各不相同。比如说，有些浏览器会把 title 属性的内容显示为弹出式的提示框，另一些浏览器则会把它们显示在状态栏里。有些浏览器会把 alt 属性的内容显示为弹出式的提示框，但因为弹出式的提示框只在鼠标悬停在有关元素之上时才会出现，所以人们往往还得使用另外一个属性来为图片再提供一个随时可见的描述。

在浏览器能否以及如何显示各有关属性的问题上，目前的普遍现状是由浏览器自行决定。但我要在这里要告诉大家，DOM 技术可以让我们把这种控制权重新掌握在自己的手里。

在这一章里，我将向大家展示几种利用 DOM 技术动态地收集和/或创建一些有用的辅助信息，并把它们动态地呈现在网页上的基本思路。你们将最先看到的前两个例子采用了同样的基本思路，它们是：

(1) 把隐藏在属性里的信息检索出来。
(2) 把这些信息动态地创建为一些 HTML 内容。
(3) 把这些 HTML 内容插入到文档里。

以上步骤与利用DOM去新建一些HTML内容的情况是有区别的。在本章的例子里，所谓"新"内容其实并不新——它们已经存在于 HTML 文档中，而我将利用 JavaScript 语言和 DOM 把它们提取出来并以另外一种结构去呈现它们。

8.3 原始内容

和往常一样，编写任何网页的出发点都是获得必要的内容。

我将把下面这段文字作为我的出发点：

```
What is the Document Object Model?
The W3C defines the DOM as:
A platform- and language-neutral interface that will allow programs
and scripts to dynamically access and update the
content, structure  and style of documents.
It is an API that can be used to navigate HTML and XML documents.
```

下面是我给上面这段文字加上了一些(X)HTML 标记后得到的 HTML 内容：

```
<h1>What is the Document Object Model?</h1>
<p>
The <abbr title="World Wide Web Consortium">W3C</abbr> defines
➥the <abbr title="Document Object Model">DOM</abbr> as:
</p>
<blockquote cite="http://www.w3.org/DOM/">
  <p>
A platform- and language-neutral interface that will allow programs
➥and scripts to dynamically access and update the
➥content, structure and style of documents.
```

```
  </p>
</blockquote>
<p>
It is an <abbr title="Application Programming Interface">API</abbr>
➥that can be used to navigate <abbr title="HyperText Markup Language">
➥HTML</abbr> and <abbr title="eXtensible Markup Language">XML
➥</abbr> documents.
</p>
```

这段 HTML 内容包含着大量的缩略词语，我已经用<abbr>标签把它们都标识出来了。

注意　<abbr>标签与<acronym>标签之间的区别是很多网页设计老手都说不清楚的问题。<abbr>标签的含义是“缩略语”（源自英文单词abbreviation），它是对单词或短语的简写形式的统称。<acronym>标签的含义是“字头缩写词”（源自英文单词acronym），它特指单词形式的缩略语。比如说，如果你把DOM念成一个单词dom，它就是一个字头缩写；如果你把它念成三个字母D-O-M，它就不是一个字头缩写词——但它仍是一个缩略语。换句话说，所有的字头缩写词都是缩略语，但并非所有的缩略语都是字头缩写词。在XHTML的未来版本里，<acronym>标签很可能会逐步淡出人们的视线。

现在，我已经把一段原始内容标记成了一段 HTML 内容，我将把这段 HTML 内容放到一份 (X)HTML 文档的上下文里。具体地说，我将先把这段 HTML 内容放入一对<body>标签，再把这个 body 元素和一个配套的 head 元素放入一对<html>标签。

选用 HTML 或是 XHTML

选用 HTML 或 XHTML 是你的自由——但这里必须遵守这样一个原则：不管你选用的是哪种文档类型，你使用的标记必须与做出的 doctype 声明保持一致。就个人而言，我更喜欢使用 XHTML。它对允许使用的标记有着更严格的要求，而这可以督促我编写出更加严谨清晰的文档。

比如说，在写出标签和属性时，HTML 既允许使用大写字母（比如<P>），也允许使用小写字母（比如<p>）；XHTML 却要求所有的标签和属性必须用小写字母写出。

HTML 往往允许我们省略一些结束标签；比如说，你可以省略</p>和
标签。这在表面看来似乎是一种方便和灵活，但从长远看却是一种隐患：万一你们的文档在浏览器里的呈现效果不符合预期，缺东少西的标记会令追查问题根源的工作变得更加困难。如果你们使用的是一种要求更为严格的 doctype，很多问题从一开始就可以避免，即使出了问题，追查其根源的工作也会容易一些。

如果你们决定选用 XHTML，就一定要把所有的结束标签都写齐全——这对诸如<img>和
之类的孤立元素也不例外：它们必须有像<img/>和
这样带有反斜杠字符的结束标签（注意，为了与早期的浏览器保持向后兼容，你们应该在反斜杠字符的前面保留一个空格）。

我们可以把 XHTML 看做是符合 XML 规范的 HTML。从技术上讲，Web 浏览器应该把 XHTML 文档当作 XML 文档而不是 HTML 文档来对待。在实际工作中，如果你想让 Web 浏览器把 XHTML 文档当作 XML 文档来对待，就必须在 XHTML 文档的头部送出正确的 MIME 类型，即 application/xhtml+xml。不过，因为有些浏览器不理解这种 MIME 类型，所以在送出这种 MIME 类型之前应该先在服务器端对客户端的浏览器进行一些嗅探以确保无疑。

如果你正在使用有着正确 MIME 类型的 XHTML，请务必记住：某些 HTML-DOM 方法和属性（例如 document.write()方法和 innerHTML 属性）不能在这种上下文里工作。但核心的 DOM 方法（DOM Core）没有这个缺陷，它们仍可以正常工作。事实上，凡是合法的 XML 文档都可以用核心的 DOM 方法来处理，它们并非只能用来处理 XHTML 文档。

即使你决定选用 HTML，你仍可以把 HTML 内容写得严谨和清晰。你可以坚持使用小写字母来写出所有的标签和属性。你可以把允许省略的</p>和
等标签增补齐全。只要注意了这些细节，你的 HTML 文档就与 XHTML 只有很小的差距了。

但请记住，HTML 迟早会被淘汰，而你迟早需要迁移到 XHTML 上去。因此，如果你还没有开始使用 XHTML，我希望你至少应该考虑去学习它。

无论你决定选用 HTML 还是选用 HTML，重要的是为你的文档选定一种 doctype 类型，并遵守与之相关的原则和规定。

8.4 XHTML 文档

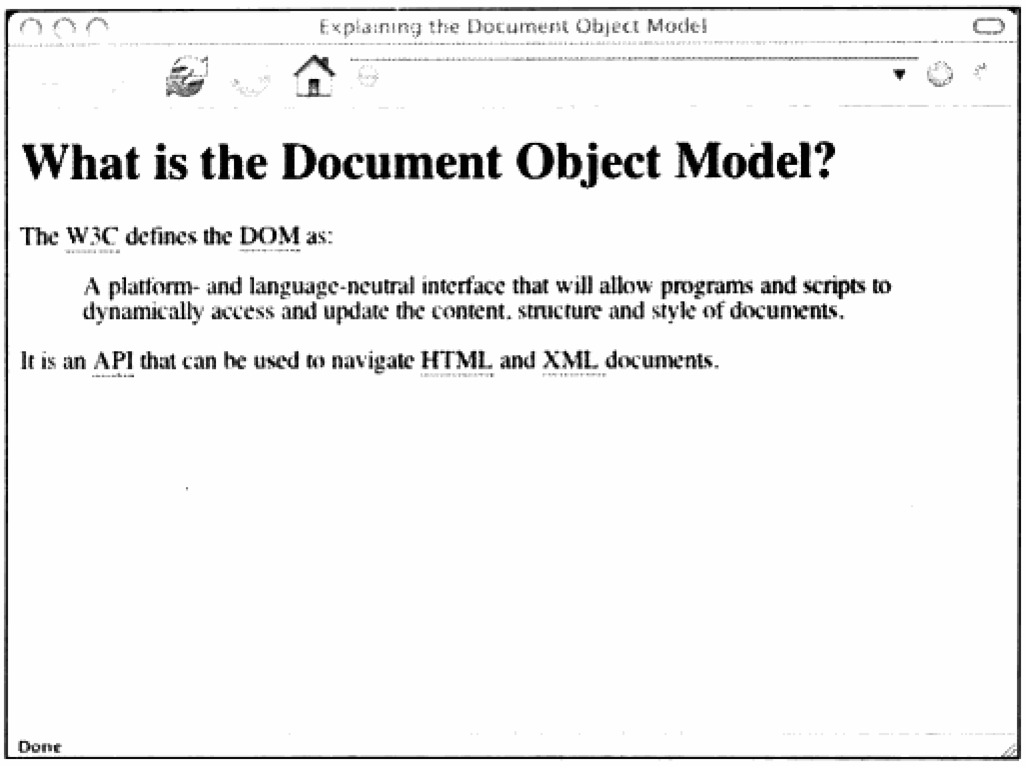
下面是把上面那段原始内容标记为一份完备的 XHTML 文档后的样子：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <meta http-equiv="content-type" content="text/html;
➥charset=utf-8" />
    <title>Explaining the Document Object Model</title>
  </head>
  <body>
    <h1>What is the Document Object Model?</h1>
    <p>
The <abbr title="World Wide Web Consortium">W3C</abbr> defines
➥the <abbr title="Document Object Model">DOM</abbr> as:
    </p>
    <blockquote cite="http://www.w3.org/DOM/">
      <p>
A platform- and language-neutral interface that will allow programs
➥and scripts to dynamically access and update the
➥content, structure and style of documents.
      </p>
```

```
    </blockquote>
    <p>
It is an <abbr title="Application Programming Interface">API</abbr>
➥that can be used to navigate <abbr title="HyperText Markup Language">
➥HTML</abbr> and <abbr title="eXtensible Markup Language">XML
➥</abbr> documents.
    </p>
  </body>
</html>
```

我决定把这个页面存为 explanation.html 文件。

如果你们在 Web 浏览器里加载这个页面，就可以看到浏览器是如何显示那些 HTML 内容的。



8.5 CSS

有些浏览器会把文档中的缩略词语（<abbr>标签）显示为带有下划线或下划点的文本，另一些浏览器则会把缩略词语显示为斜体字。

虽然我还未给 explanation.html 文档配上任何样式表，但样式显然已经在起作用了。这是因为每种浏览器都有一些自己的默认样式。

我们可以用自己的样式表来取代浏览器的默认样式。请看下面这个例子：

```
body {
  font-family: "Helvetica","Arial",sans-serif;
  font-size: 10pt;
}
abbr {
  text-decoration: none;
```

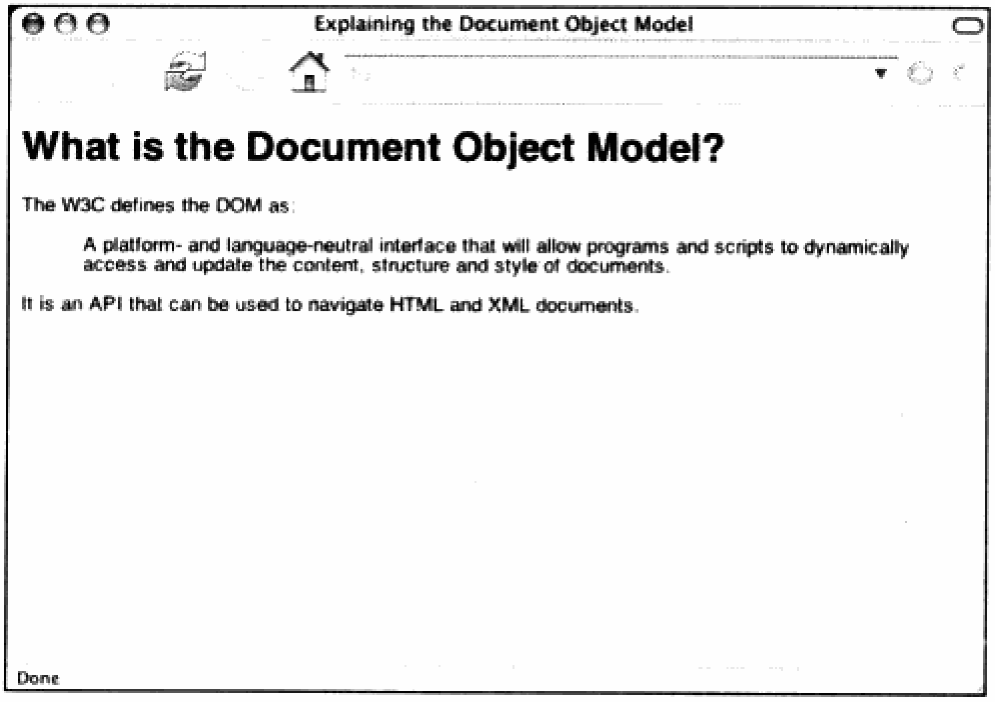
```
    border: 0;
    font-style: normal;
  }
```

我决定把这个样式表保存为 typography.css 文件，并将其放到子目录 styles 里去。

现在，我将在 explanation.html 文档的<head>部分增加一条如下所示的语句：

```
<link rel="stylesheet" type="text/css" media="screen"
➥href="styles/typography.css" />
```

如果把 explanation.html 文档加载到一个 Web 浏览器里，就可以看到一些样式方面的差别。这份文档的内容现在将被显示为另外一种字体，其中的缩略词语已经无特别之处了。



8.6 JavaScript 代码

缩略词语（<abbr>标签）的 title 属性在浏览器里一般是不可见的。有些浏览器会在鼠标指针悬停在缩略词语上方时，把它的 title 属性显示为一个弹出式的提示框。就像浏览器所使用的默认样式一样，浏览器对缩略词语的默认呈现行为也是各有各的做法。

就像我们可以用自己的 CSS 样式表去取代浏览器所使用的默认样式那样，我们可以用 DOM 脚本去改变浏览器的默认行为。

8.6.1 显示“缩略词语表”

在 explanation.html 文档里有很多<abbr>标签，我想把这些<abbr>标签的 title 属性值收集起来，并将其集中显示在页面里。用一个 HTML 定义表（definition list）元素（<dl>）来显示这些<abbr>标签所包含的值（<abbr>标签的节点值）和<abbr>标签的 title 属性值是再合适不过的了。下面是我希望得到的定义表元素：

```
<dl>
  <dt>W3C</dt>
  <dd>World Wide Web Consortium</dd>
  <dt>DOM</dt>
  <dd>Document Object Model</dd>
  <dt>API</dt>
  <dd>Application Programming Interface</dd>
  <dt>HTML</dt>
  <dd>HyperText Markup Language</dd>
  <dt>XML</dt>
  <dd>eXtensible Markup Language</dd>
</dl>
```

我将使用一些 DOM 方法和属性来创建这个定义表，具体步骤如下：

(1) 遍历这份文档中的所有 abbr 元素。
(2) 把每个 abbr 元素的 title 属性值提取出来。
(3) 把每个 abbr 元素的值提取出来。
(4) 创建一个“定义表”元素（即 dl 元素）。
(5) 遍历刚才提取出来的 title 属性值和 abbr 元素的值。
(6) 创建一个“定义标题”元素（即 dt 元素）。
(7) 把 abbr 元素的值插入这个 dt 元素。
(8) 创建一个“定义描述”元素（即 dd 元素）。
(9) 把 title 属性值插入这个 dd 元素。
(10) 把 dt 元素追加到第 4 步创建的 dl 元素上。
(11) 把 dd 元素追加到第 4 步创建的 dl 元素上。
(12) 把 dl 元素追加到 explanation.html 文档的 body 元素上。

我将编写一个函数来完成以上步骤，并把这个函数命名为 displayAbbreviations()。然后将其保存到 displayAbbreviations.js 文件中并存放到子目录 scripts 里去。

1. 编写displayAbbreviations()函数

第一步是定义这个函数。因为它不需要任何参数，所以函数名后面的圆括号将是空的：

```
function displayAbbreviations() {
```

开始遍历这份文档里的所有 abbr 元素之前，我必须先把它们找出来。这可以用 getElementsByTagName()方法轻松完成：只需把 abbr 传递给这个方法，它就会返回一个包含着这份文档里的所有 abbr 元素节点的节点集合。正如前面的有关章节里提到的那样，节点集合就是一个由节点构成的数组。我将把这个数组保存到变量 abbreviations 里去：

```
var abbreviations = document.getElementsByTagName("abbr");
```

现在，可以开始遍历 abbreviations 数组了。但在这之前，我还想先进行一些测试和检查。

我知道在这份特定的文档里有一些缩略词语，但并非所有的文档都会如此。如果我们想让这个函数还能适用于其他文档，就应该趁现在这个机会先去检查一下当前文档是不是包含有缩略词语。

我们知道，abbreviations 数组的 length 属性值就是这个文档里的缩略词语的个数。如果 abbreviations.length 小于 1，就说明这个文档里无缩略词语。如果真是这样，这个函数就应该立刻停止执行并返回一个布尔值 false：

```
if (abbreviations.length < 1) return false;
```

如果文档里没有 abbr 元素，这个函数将就此结束。

下一步是从每个 abbr 元素提取信息。我需要把<abbr>标签里的文本以及每个 title 属性的值提取出来。当我们需要把像这样的一系列数据值保存起来时，理想的存储媒介将是一个数组。

下面这条语句定义了一个名为 defs 的新数组：

```
var defs = new Array();
```

下一步是遍历 abbreviations 数组：

```
for (var i=0; i<abbreviations.length; i++) {
```

为了提取出对当前缩略词语的释义文字，我需要对 title 属性使用 getAttribute()方法并把提取出来的值保存到变量 definition 里：

```
var definition = abbreviations[i].getAttribute("title");
```

提取<abbr>标签里的缩略词语需要用到 nodeValue 属性。需要提取的是 abbr 元素里的文本节点的值。在 explanation.html 文档中的每个 abbr 元素里，文本节点都是这个元素内部的第一个（也是仅有的一个）节点。换句话说，那个文本节点是 abbr 元素节点的第一个子节点：

```
abbreviations[i].firstChild
```

不过，那个文本也有可能会嵌套在其他的元素里。请看下面这条 HTML 语句：

```
<abbr title="Document Object Model"><em>DOM</em></abbr>
```

此时，abbr 元素节点的第一个子节点将是 em 元素节点，文本节点是 em 元素的子节点。因此，与其使用 firstChild 属性，不如使用 lastChild 属性更稳妥：

```
abbreviations[i].lastChild
```

下面这条语句将提取出这个文本节点的 nodeValue 属性并把它赋值给变量 key：

```
var key = abbreviations[i].lastChild.nodeValue;
```

现在有两个变量了：definition 和 key。这两个变量的值就是我想保存到 defs 数组里的内容。我将使用如下所示的语句来保存它们：把其中之一用作数组元素的下标（键字），另一个用作数组元素的值：

```
defs[key] = definition;
```

defs 数组中的第一个元素的下标是“W3C”，取值是“World Wide Web Consortium”；defs 数组中的第二个元素的下标是“DOM”，取值是“Document Object Model”；依此类推。

下面是这个 for 循环的完整代码：

```
for (var i=0; i<abbreviations.length; i++) {
  var definition = abbreviations[i].getAttribute("title");
  var key = abbreviations[i].lastChild.nodeValue;
  defs[key] = definition;
}
```

如果想让这个函数更容易阅读和理解，我建议你把 abbreviations[i]的值——你在本次循环里正在对之进行处理的那个 abbreviations 数组元素——赋值给变量 current_abbr：

```
for (var i=0; i<abbreviations.length; i++) {
  var current_abbr = abbreviations[i];
  var definition = current_abbr.getAttribute("title");
  var key = current_abbr.lastChild.nodeValue;
  defs[key] = definition;
}
```

如果你觉得 current_abbr 变量可以帮助你更好地理解这段代码，那就把它留在那里好了。额外增加一条这样的语句并不会增加多少开销，但回报却往往很大。

从理论上讲，我完全可以把整个循环体写成一条如下所示语句，但那会让代码非常难以阅读和理解：

```
for (var i=0; i<abbreviations.length; i++) {
  defs[abbreviations[i].lastChild.nodeValue] =
➥abbreviations[i].getAttribute("title");
 }
```

在编写 JavaScript 代码时，许多操作都有一种以上的实现办法。就拿上面这个 for 循环来说吧，我已经给出了它的三种不同的写法，你们可以从中选出一种你们认为最好的写法用在脚本里。请记住，如果你在编写某些代码时就觉得它们不容易理解，等你日后再去阅读这些代码的时候就极可能会更看不明白。

现在，我已经把那些缩略词语及其解释保存到了 defs 数组里。我将利用这个数组去创建一段 HTML 内容以把那些缩略词语及其解释显示在页面上。

2. 创建HTML内容

“定义表”（<dl>）是把一些缩略词语及其解释生成为一个清单的最佳选择。定义表（<dl>）由一系列“定义标题”（<dt>）和相应的“定义描述”（<dd>）构成：

```
<dl>
  <dt>Title 1</dt>
  <dd>Description 1</dd>
  <dt>Title 2</dt>
```

```
  <dd>Description 2</dd>
</dl>
```

我将用 createElement()方法去创建这个定义表，并把新创建的 dl 元素赋值给变量 dlist：

```
var dlist = document.createElement("dl");
```

由上面这条语句创建出来的 dl 元素只是 JavaScript 世界里的一个孤儿。我将通过引用 dlist 变量把它插入到 explanation.html 文档里去。

我现在需要再去编写一个循环语句，并在其中对刚创建的 defs 数组进行遍历。我决定把它也编写为一个 for 循环，但这与我在前面编写的那个 for 循环有点儿不一样。

- for/in循环

for/in 循环的独特之处是，它可以把某个数组的下标（键字）临时地赋值给一个变量：

for (variable in array)

在 for/in 循环进入第一次循环时，变量 *variable* 将被赋值为数组 *array* 的第一个元素的下标值；在进入第二次循环时，变量 *variable* 将被赋值为数组 *array* 的第二个元素的下标值；依此类推，直到遍历完数组 *array* 里的所有元素为止。

既然如此，我决定使用一个如下所示的 for/in 循环去遍历 defs 关联数组：

```
for (key in defs) {
```

上面这行代码的含义是"对于 defs 关联数组里的每个键字，把它的值赋给变量 key"。在接下来的循环体部分，变量 key 可以像其他变量那样使用。具体到这个例子，因为变量 key 的值是 defs 数组在本次循环里的键字，所以可以利用它把相应的数组元素值检索出来：

```
var definition = defs[key];
```

在这个 for/in 循环的第一次循环里，变量 key 的值是"W3C"，变量 definition 的值是"World Wide Web Consortium"；在第二次循环里，变量 key 的值是"DOM"，变量 definition 的值是"Document Object Model"。

在每次循环里，都需要创建一个 dt 元素和一个 dd 元素。还需要创建两个文本节点并把它们分别插入到新创建的 dt 和 dd 元素。

我决定先创建那个 dt 元素：

```
var dtitle = document.createElement("dt");
```

然后用变量 key 的值去创建一个文本节点：

```
var dtitle_text = document.createTextNode(key);
```

已经创建了两个节点。我把新创建的元素节点赋值给变量 dtitle。把新创建的文本节点赋值给变量 dtitle_text。我将使用 appendChild()方法把 dtitle_text 文本节点插入到 dtitle 元素节点：

```
dtitle.appendChild(dtitle_text);
```

接下来是创建 dd 元素：

```
var ddesc = document.createElement("dd");
```

然后用变量 definition 的值去创建一个文本节点：

```
var ddesc_text = document.createTextNode(definition);
```

再用 appendChild()方法把 ddesc_text 文本节点插入到 ddesc 元素节点：

```
ddesc.appendChild(ddesc_text);
```

现在，我有了两个元素节点：dtitle 和 ddesc。这两个元素节点分别包含着文本节点 dtitle_text 和 ddesc_text。

在结束循环之前，还需要把新创建的 dt 和 dd 元素追加到我稍早创建的 dl 元素上——我已经把这个 dl 元素赋值给了变量 dlist：

```
dlist.appendChild(dtitle);
dlist.appendChild(ddesc);
```

下面是这个 for/in 循环的完整代码：

```
for (key in defs) {
  var definition = defs[key];
  var dtitle = document.createElement("dt");
  var dtitle_text = document.createTextNode(key);
  dtitle.appendChild(dtitle_text);
  var ddesc = document.createElement("dd");
  var ddesc_text = document.createTextNode(definition);
  ddesc.appendChild(ddesc_text);
  dlist.appendChild(dtitle);
  dlist.appendChild(ddesc);
}
```

到了这个阶段，我的“缩略词语表”就完成了。它已经作为一个 DocumentFragment 对象存在于 JavaScript 上下文里。接下来的工作是把它插入到文档中去。

与把这个定义表突兀地插入文档的做法相比，给它加上一个描述性标题的做法应该会有更好的效果。

先创建一个 h2 元素节点：

```
var header = document.createElement("h2");
```

再创建一个内容为“Abbreviations”的文本节点：

```
var header_text = document.createTextNode("Abbreviations");
```

然后把文本节点插入元素节点：

```
header.appendChild(header_text);
```

对于结构比较复杂的文档，或许还需要借助于某个特定的 id 才能把新创建的元素插入到文档里的某个特定位置。因为 explanations.html 文档的结构并不复杂，所以我只要把新创建的元素追加到 body 标签上即可。

引用 body 标签的具体做法有两种。第一种是使用 DOM Core，即引用某给定文档的第一个（也是仅有的一个）body 标签：

```
document.getElementsByTagName("body")[0]
```

第二种做法是使用 HTML-DOM，即引用某给定文档的 body 属性：

```
document.body
```

首先，插入“缩略词语表”的标题：

```
document.body.appendChild(header);
```

然后，插入“缩略词语表”本身：

```
document.body.appendChild(dlist);
```

终于把 displayAbbreviations()函数编写出来了：

```
function displayAbbreviations() {
  var abbreviations = document.getElementsByTagName("abbr");
  if (abbreviations.length < 1) return false;
  var defs = new Array();
  for (var i=0; i<abbreviations.length; i++) {
    var current_abbr = abbreviations[i];
    var definition = current_abbr.getAttribute("title");
    var key = current_abbr.lastChild.nodeValue;
    defs[key] = definition;
  }
  var dlist = document.createElement("dl");
  for (key in defs) {
    var definition = defs[key];
    var dtitle = document.createElement("dt");
    var dtitle_text = document.createTextNode(key);
    dtitle.appendChild(dtitle_text);
    var ddesc = document.createElement("dd");
    var ddesc_text = document.createTextNode(definition);
    ddesc.appendChild(ddesc_text);
    dlist.appendChild(dtitle);
    dlist.appendChild(ddesc);
  }
  var header = document.createElement("h2");
  var header_text = document.createTextNode("Abbreviations");
  header.appendChild(header_text);
  document.body.appendChild(header);
```

```
  document.body.appendChild(dlist);
}
```

依照惯例，现在要去看看这个函数是否还有需要改进和完善的地方。

首先，在这个函数的开头部分，应该安排一些测试以确保浏览器能够理解我在这个函数里用到的 DOM 方法。我在这个函数里用到了 getElementsByTagName()、createElement()和 createTextNode()方法。我将使用以下语句去检查这几个方法是否存在：

```
if (!document.getElementsByTagName) return false;
if (!document.createElement) return false;
if (!document.createTextNode) return false;
```

当然，如果我愿意，也可以把这几项测试组合为一条语句：

```
if (!document.getElementsByTagName || !document.createElement
➥|| !document.createTextNode) return false;
```

这两种做法并无本质区别，你们可以根据自己的个人习惯加以选择。

其次，因为 displayAbbreviations()函数比较复杂，所以还应该在它的代码里加上一些注释。下面是我最终完成的 displayAbbreviations()函数：

```
function displayAbbreviations() {
  if (!document.getElementsByTagName || !document.createElement
➥|| !document.createTextNode) return false;
// get all the abbreviations
  var abbreviations = document.getElementsByTagName("abbr");
  if (abbreviations.length < 1) return false;
  var defs = new Array();
// loop through the abbreviations
  for (var i=0; i<abbreviations.length; i++) {
    var current_abbr = abbreviations[i];
    var definition = current_abbr.getAttribute("title");
    var key = current_abbr.lastChild.nodeValue;
    defs[key] = definition;
  }
// create the definition list
  var dlist = document.createElement("dl");
// loop through the definitions
  for (key in defs) {
    var definition = defs[key];
// create the definition title
    var dtitle = document.createElement("dt");
    var dtitle_text = document.createTextNode(key);
    dtitle.appendChild(dtitle_text);
// create the definition description
    var ddesc = document.createElement("dd");
    var ddesc_text = document.createTextNode(definition);
    ddesc.appendChild(ddesc_text);
```

```
// add them to the definition list
      dlist.appendChild(dtitle);
      dlist.appendChild(ddesc);
    }
// create a headline
    var header = document.createElement("h2");
    var header_text = document.createTextNode("Abbreviations");
    header.appendChild(header_text);
// add the headline to the body
    document.body.appendChild(header);
// add the definition list to the body
    document.body.appendChild(dlist);
}
```

但事情还没有结束——还需要安排这个函数在页面加载时被调用。可以通过 window.onload 事件来做到这一点：

```
window.onload = displayAbbreviations;
```

为了让自己日后能够方便地把多个事件添加到 window.onload 处理函数上，我决定使用 addLoadEvent()函数来完成这一工作：首先，编写 addLoadEvent()函数并把它保存为一个新的 JavaScript 脚本文件——将新文件命名为 addLoadEvent.js 并把它存入 scripts 文件夹：

```
function addLoadEvent(func) {
  var oldonload = window.onload;
  if (typeof window.onload != 'function') {
    window.onload = func;
  } else {
    window.onload = function() {
      oldonload();
      func();
    }
  }
}
```

然后，把下面这条语句添加到 displayAbbreviations.js 文件里：

```
addLoadEvent(displayAbbreviations);
```

现在，JavaScript 脚本文件都已经准备好了。接下来，为了调用这两个 JavaScript 脚本文件，我还需要在 explanation.html 文件的<head>部分添加一些<script>标签，如下所示：

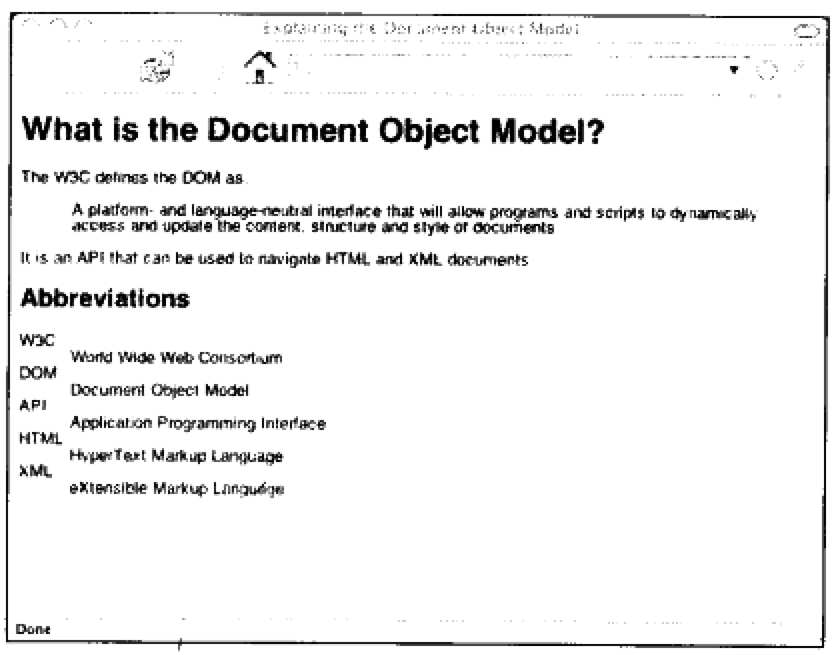
```
<script type="text/javascript" src="scripts/addLoadEvent.js"></script>
<script type="text/javascript"
➥src="scripts/displayAbbreviations.js"></script>
```

下面是最终完成的 explanation.html 文件：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
```

```
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>Explaining the Document Object Model</title>
    <link rel="stylesheet" type="text/css" media="screen"
➥href="styles/typography.css" />
    <script type="text/javascript" src="scripts/addLoadEvent.js">
    </script>
    <script type="text/javascript" src="scripts/displayAbbreviations.js">
    </script>
  </head>
<body>
    <h1>What is the Document Object Model?</h1>
    <p>
The <abbr title="World Wide Web Consortium">W3C</abbr> defines
➥the <abbr title="Document Object Model">DOM</abbr> as:
    </p>
    <blockquote cite="http://www.w3.org/DOM/">
      <p>
A platform- and language-neutral interface that will allow programs
➥and scripts to dynamically access and update the
➥content, structure and style of documents.
      </p>
    </blockquote>
    <p>
It is an <abbr title="Application Programming Interface">API</abbr>
➥that can be used to navigate <abbr title="HyperText Markup Language">
➥HTML</abbr> and <abbr title="eXtensible Markup Language">XML
</abbr> documents.
    </p>
  </body>
</html>
```

现在，把 explanation.html 文件加载到一个 Web 浏览器里就可以看到 displayAbbreviations() 函数的效果了。



3. 埋藏在浏览器里的"地雷"

至此，我一直在避免提到任何特定的浏览器。只要使用的浏览器支持 DOM，则此前见到过的脚本就都可以正常工作。可是，我刚刚编写的 displayAbbreviations()函数却是这一规律的一个例外。

displayAbbreviations()函数工作得确实不错，但这必须以你们使用的浏览器不是 IE 浏览器的 Windows 版本为前提：如果把 explanation.html 文件加载到 IE 浏览器里，你们不仅不会看到一个"缩略词语表"，还极有可能会看到一条 JavaScript 出错消息。

你们肯定会对 IE 浏览器的这种行为感到不解：我们已经在 displayAbbreviations()函数的开头部分加上了必要的对象探测语句，以确保只有那些支持 DOM 的浏览器才会去执行 DOM 代码，IE 浏览器对 getElementsByTagName()和 getElementById()方法的支持也毋庸置疑，为什么还会出现这样的问题呢？

事情还要从我在本书第一章里提到的浏览器大战说起。在那场大战中，网景公司和微软公司曾把<abbr>和<acronym>标签当做它们的武器之一。在竞争最激烈时，微软决定不在自己的浏览器里实现 abbr 元素。

那场浏览器大战早已烟消云散，最终的结果是微软打败了网景。作为那场战争的后遗症之一，出自微软的 IE 浏览器直到现在仍不支持 abbr 元素，它在加载一个页面并为之创建 DOM 时会拒绝把<abbr>标签识别为一个元素节点！我们的 explanation.html 文档之所以会在 IE 浏览器里出问题，是因为 displayAbbreviations()函数试图从一些 abbr 元素节点那里提取属性节点和文本节点，而 IE 浏览器却拒绝承认那些 abbr 节点的"元素"地位。

我们意外地踏上了一颗在一场早已结束的战争中埋藏下来的"地雷"！

可供选择的解决方案有两种：

(1) 把 explanation.html 文档里的 abbr 元素统一替换为 acronym 元素。
(2) 在 displayAbbreviations()函数里为它遇到 IE 浏览器的情况预留一条退路。

我对第一种解决方案不感兴趣，因为我不想为了迁就一种顽固不化的浏览器而"牺牲"一大批语义正确的 HTML 代码。

第二种解决方案相当容易实现。我只须在 displayAbbreviations()函数里增加一两条语句，就可以让 IE 以及所有不能识别 abbr 元素的其他浏览器尽早离开这个函数了。

首先，我将在负责从各个 abbr 元素提取其 title 属性值和文本节点值的 for 循环里添加一条语句：

```
for (var i=0; i<abbreviations.length; i++) {
  var current_abbr = abbreviations[i];
  if (current_abbr.childNodes.length < 1) continue;
  var definition = current_abbr.getAttribute("title");
```

```
    var key = current_abbr.lastChild.nodeValue;
    defs[key] = definition;
  }
```

这条新增语句的含义是："如果当前元素没有子节点，就立刻开始下一次循环"。因为 IE 浏览器在统计 abbr 元素的子节点个数时总是会"正确地"返回一个错误的值——零，所以这条新语句会让 IE 浏览器不再继续执行这个循环中的后续代码。

此外，当 IE 浏览器执行到 displayAbbreviations()函数中负责创建"缩略词语表"的那个 for 循环时，因为 defs 数组是空的，所以它将不会创建出任何 dt 和 dd 元素。我将在那个 for 循环的后面添加这样一条语句：如果对应于"缩略词语表"的那个 dl 元素没有任何子节点，则立刻退出 displayAbbreviations()函数：

```
// create the definition list
var dlist = document.createElement("dl");
// loop through the definitions
for (key in defs) {
  var definition = defs[key];
// create the definition title
  var dtitle = document.createElement("dt");
  var dtitle_text = document.createTextNode(key);
  dtitle.appendChild(dtitle_text);
// create the definition description
  var ddesc = document.createElement("dd");
  var ddesc_text = document.createTextNode(definition);
  ddesc.appendChild(ddesc_text);
// add them to the definition list
  dlist.appendChild(dtitle);
  dlist.appendChild(ddesc);
}
if (dlist.childNodes.length < 1) return false;
```

请注意，新添加的这条 if 语句又一次违背了结构化程序设计原则——它等于是在 displayAbbreviations()函数的中间增加了一个出口点。但我这么做是有理由的：这应该是既可以解决 IE 浏览器的问题，又不需要对现有的函数代码大动干戈的最简单的办法了。

下面是 displayAbbreviations()函数在我对它进行了上述改进之后的代码清单：

```
function displayAbbreviations() {
  if (!document.getElementsByTagName || !document.createElement
➥|| !document.createTextNode) return false;
// get all the abbreviations
  var abbreviations = document.getElementsByTagName("abbr");
  if (abbreviations.length < 1) return false;
  var defs = new Array();
// loop through the abbreviations
  for (var i=0; i<abbreviations.length; i++) {
    var current_abbr = abbreviations[i];
```

```
        if (current_abbr.childNodes.length < 1) continue;
        var definition = current_abbr.getAttribute("title");
        var key = current_abbr.lastChild.nodeValue;
        defs[key] = definition;
      }
    // create the definition list
      var dlist = document.createElement("dl");
    // loop through the definitions
      for (key in defs) {
        var definition = defs[key];
    // create the definition title
        var dtitle = document.createElement("dt");
        var dtitle_text = document.createTextNode(key);
        dtitle.appendChild(dtitle_text);
    // create the definition description
        var ddesc = document.createElement("dd");
        var ddesc_text = document.createTextNode(definition);
        ddesc.appendChild(ddesc_text);
    // add them to the definition list
        dlist.appendChild(dtitle);
        dlist.appendChild(ddesc);
      }
      if (dlist.childNodes.length < 1) return false;
    // create a headline
      var header = document.createElement("h2");
      var header_text = document.createTextNode("Abbreviations");
      header.appendChild(header_text);
    // add the headline to the body
      document.body.appendChild(header);
    // add the definition list to the body
      document.body.appendChild(dlist);
    }
```

那两条新语句将确保我的 explanation.html 文档就算遇到那些不理解 abbr 元素的浏览器也不会出问题。它们就像是一条保险绳，其作用与脚本开头部分的对象探测语句很相似。

我希望大家能够通过上面这个例子明白一个道理：即使某种特定的浏览器会引起问题，也没有必要使用浏览器嗅探代码。对浏览器的名称和版本号进行嗅探的办法很难做到面面俱到，而且往往会导致非常复杂难解的代码。

我们已经成功地排除了一颗在过去的浏览器大战中遗留下来的“地雷”。如果说有什么教训的话，那就是它可以让我们深刻地体会到标准的重要性。仅仅因为 IE 浏览器不支持 abbr 元素，就使得一大批用户没有机会看到一个自动生成的“缩略词语表”的事实让我感到很遗憾，但值得庆幸的是，那些用户仍能看到页面上的核心内容。在这个例子里，“缩略词语表”只是一种很好的补充，它还算不上是页面必不可少的组成部分。如果它真的是必不可少，我们从一开始就应该把它包括在 HTML 文档里。

你们可以从本书在 Friends of ED 网站（http://friendsofed.com/）上的下载主页里找到 displayAbbreviations()函数的最终版本。

8.6.2 显示“文献来源链接表”

displayAbbreviations()函数是一个充实文档内容的好例子（至少对那些不是 IE 的浏览器来说是如此）。它从文档结构的现有内容里提取出了一些信息并以一种清晰可见的方式把它们显示出来。那些原本包含在 abbr 标签的 title 属性里的信息现在直接出现在了浏览器里。

现在，请大家仔细看一下 explanation.html 文档中的下面这段 HTML 内容：

```
<blockquote cite="http://www.w3.org/DOM/">
  <p>
A platform- and language-neutral interface that will allow programs
➥and scripts to dynamically access and update the
➥content, structure and style of documents.
  </p>
</blockquote>
```

blockquote 元素包含着一个属性 cite。这个属性是可选的，它的基本用途是给出一个用来告诉人们 blockquote 元素的内容是来自何方的 URL 地址。

从理论上讲，这是一个把文献资料与相关网页链接起来的好办法；但在实践中，浏览器会完全忽视 cite 属性的存在。虽然信息就在那里，但用户却无法看到它们。利用 JavaScript 语言和 DOM，我们完全可以把那些信息收集起来并以一种更有意义的方式把它们显示在网页上。

我将按照以下步骤去收集和显示 blockquote 元素的 cite 属性所包含的信息（以 explanation.html 文档为例）：

(1) 遍历这个文档里所有 blockquote 元素。
(2) 从 blockquote 元素提取出 cite 属性的值。
(3) 创建一个标识文本是“source”的链接。
(4) 把这个链接赋值为 blockquote 元素的 cite 属性值。
(5) 把这个链接插入到文献节选的末尾。

下面，我将根据上述步骤编写一个名为“displayCitations”的 JavaScript 函数，并把它存放到文件 displayCitations.js 里。

编写displayCitations()函数

我们先来看看如何编写这个函数。因为它不需要任何参数，所以函数名后面的圆括号将是空的：

```
function displayCitations() {
```

第一步是把文档里的所有 blockquote 元素找出来。我将使用 getElementsByTagName()方法来完成这项查找工作并把找到的节点集合保存为变量 quotes：

```
var quotes = document.getElementsByTagName("blockquote");
```

接下来是遍历这个集合：

```
for (var i=0; i<quotes.length; i ++) {
```

在这个循环里，我只对有 cite 属性的文献节选感兴趣。我将用一个简单的测试去检查本次循环中的当前文献节选有没有这个属性。

接受这个测试的是节点集合 quotes 中的当前元素（即 quotes[i]）。为了进行这一测试，需要把这个节点作为参数传递给 getAttribute()方法：如果 getAttribute("cite")的结果是 true，就说明这个节点有 cite 属性；如果!getAttribute("cite")的结果是 true，就说明这个节点没有 cite 属性。如果是后一种情况，我将使用 JavaScript 关键字 continue 立刻跳转到下一次循环的开始，不再继续执行本次循环中的后续语句：

```
if (!quotes[i].getAttribute("cite")) {
   continue;
}
```

也可以把这条语句写成下面这样：

```
if (!quotes[i].getAttribute("cite")) continue;
```

接下来的语句将只在当前 blockquote 元素节点有 cite 属性节点的情况下才会执行。

首先，提取当前 blockquote 元素的 cite 属性值并把它存入变量 url：

```
var url = quotes[i].getAttribute("cite");
```

下一步是确定应该把“文献来源链接”放到何处。这似乎是一项非常简单的任务。

- 查找元素节点

文献节选由一段或多段文本构成，所以 blockquote 元素肯定包含着一个或多个 p 元素。我想把“文献来源链接”放在 blockquote 元素所包含的最后一个子元素节点的后面。最容易想到的办法是，利用当前 blockquote 元素的 lastChild 属性来完成这一操作：

```
quotes[i].lastChild
```

可是，如果真这样做了，就会遇到一个问题。请大家再仔细看看那段 HTML 原始内容：

```
<blockquote cite="http://www.w3.org/DOM/">
  <p>
A platform- and language-neutral interface that will allow programs
➥and scripts to dynamically access and update the
➥content, structure and style of documents.
  </p>
</blockquote>
```

乍看起来，blockquote 元素的最后一个子节点应该是那个 p 元素，而这意味着 lastChild 属性的返回值将是一个 p 元素节点。可是，事实却并不一定如此。

那个 p 节点的确是 blockquote 元素的最后一个元素节点。但在</p>标签和</blockquote>标签之间还存在着一个换行符。有些浏览器会把这个换行符解释为一个文本节点。这样一来，blockquote 元素节点的 lastChild 属性就将是一个文本节点而不是那个 p 元素节点。

注意　在编写DOM脚本时，想当然地认为某个节点肯定是一个元素节点是一种相当常见的错误。如果没有百分之百的把握，就一定要去检查nodeType属性值。有些DOM方法只能用于元素节点；如果你把它们用在了文本节点身上，就会“创造”出一个错误。

DOM 已经为我们提供了一个非常有用的 lastChild 属性，如果它能再为我们提供一个 lastChildElement 属性就更好了——但令人遗憾的是它没有。还好，我们完全可以利用现有的 DOM 方法和属性编写出能够完全这项任务的语句来。

我们可以把包含在当前 blockquote 元素里的所有元素节点找出来。如果把通配符“*”作为参数传递给 getElementsByTagName()方法，它就会把所有的元素——不管具体的 HTML 标记是什么，返回给我们：

```
var quoteChildren = quotes[i].getElementsByTagName("*");
```

变量 quoteElements 是一个数组，它包含着当前 blockquote 元素（即 quotes[i]）所包含的全体元素节点。

现在，blockquote 元素所包含的最后一个元素节点将对应着 quoteElements 数组中的最后一个元素。数组中的最后一个元素的下标等于数组的长度减去 1，因为数组的下标从零开始。记住，数组中的最后一个元素的下标不等于数组的长度，它等于数组的长度减去 1：

```
var elem = quoteChildren[quoteChildren.length - 1];
```

现在，变量 elem 对应着 blockquote 元素所包含的最后一个元素节点。

回到我正在 displayCitations()函数里编写的那个循环，下面是已经编写出来的代码：

```
for (var i=0; i<quotes.length; i++) {
  if (!quotes[i].getAttribute("cite")) continue;
  var url = quotes[i].getAttribute("cite");
  var quoteChildren = quotes[i].getElementsByTagName('*');
  var elem = quoteChildren[quoteChildren.length - 1];
```

与其假设 quoteChildren 变量肯定是一个元素节点数组，我决定增加一项测试来检查它的长度是否小于 1。如果是，就用关键字 continue 立刻退出本次循环：

```
for (var i=0; i<quotes.length; i++) {
  if (!quotes[i].getAttribute("cite")) continue;
  var url = quotes[i].getAttribute("cite");
  var quoteChildren = quotes[i].getElementsByTagName('*');
  if (quoteChildren.length < 1) continue;
  var elem = quoteChildren[quoteChildren.length - 1];
```

我已经把创建一个链接所需要的东西全准备好了：变量 url 包含着将成为那个链接的 href 属性值的信息；elem 变量包含着将成为那个链接在文档中的插入位置的节点。

用 createElement()方法创建一个“链接”元素：

```
var link = document.createElement("a");
```

接下来，我还需要为那个新链接创建一条标识文本。用 createTextNode()方法创建一个内容为“source”的文本节点：

```
var link_text = document.createTextNode("source");
```

现在，变量 link 包含着新创建的 a 元素，变量 link_text 包含着新创建的文本节点。

用 appendChild()方法把新的文本节点插入新链接：

```
link.appendChild(link_text);
```

把 href 属性添加给新链接。用 setAttribute()方法把它设置为变量 url 的值：

```
link.setAttribute("href",url);
```

新链接已经创建好了。我可以就这样把它插入文档，也可以先用另一个元素把它“包装”一下再插入文档。我决定用一个 sup 元素来包装它，这将使它在浏览器里呈现出上标的效果。

创建一个 sup 元素节点并把它存入变量 superscript：

```
var superscript = document.createElement("sup");
```

把新链接放入这个 sup 元素：

```
superscript.appendChild(link);
```

现在，有了一个存在于 JavaScript 上下文中的 DocumentFragment 对象，它此时尚未被插入任何文档：

```
<sup><a href="http://www.w3.org/DOM/">source</a></sup>
```

为了把这条 HTML 语句插入文档，我将把变量 superscript 追加为变量 elem 的最后一个子节点。从效果上看，因为变量 elem 对应着 blockquote 元素目前所包含的最后一个元素节点，这个上标形式的新链接将出现在文献节选的紧后面：

```
elem.appendChild(superscript);
```

最后，先用一个右花括号结束这个 for 循环，再用一个右花括号结束整个函数。

下面是 displayCitations()函数到目前为止的代码清单：

```
function displayCitations() {
  var quotes = document.getElementsByTagName("blockquote");
  for (var i=0; i<quotes.length; i++) {
    if (!quotes[i].getAttribute("cite")) continue;
```

```
        var url = quotes[i].getAttribute("cite");
        var quoteChildren = quotes[i].getElementsByTagName('*');
        if (quoteChildren.length < 1) continue;
        var elem = quoteChildren[quoteChildren.length - 1];
        var link = document.createElement("a");
        var link_text = document.createTextNode("source");
        link.appendChild(link_text);
        link.setAttribute("href",url);
        var superscript = document.createElement("sup");
        superscript.appendChild(link);
        elem.appendChild(superscript);
      }
    }
```

依照惯例，我们来看看这个函数是否还有需要改进和完善的地方。在这个函数的开头部分增加一些测试，以确保浏览器能够理解我在这个函数里用到的 DOM 方法。为了让代码更易于理解，我还在加上了一些注释：

```
    function displayCitations() {
      if (!document.getElementsByTagName || !document.createElement
    ➥|| !document.createTextNode) return false;
    // get all the blockquotes
      var quotes = document.getElementsByTagName("blockquote");
    // loop through all the blockquotes
      for (var i=0; i<quotes.length; i++) {
    // if there is no cite attribute, continue the loop
        if (!quotes[i].getAttribute("cite")) continue;
    // store the cite attribute
        var url = quotes[i].getAttribute("cite");
    // get all the element nodes in the blockquote
        var quoteChildren = quotes[i].getElementsByTagName('*');
    // if there are no element node, continue the loop
        if (quoteChildren.length < 1) continue;
    // get the last element node in the blockquote
        var elem = quoteChildren[quoteChildren.length - 1];
    // create the markup
        var link = document.createElement("a");
        var link_text = document.createTextNode("source");
        link.appendChild(link_text);
        link.setAttribute("href",url);
        var superscript = document.createElement("sup");
        superscript.appendChild(link);
    // add the markup to the last element node in the blockquote
        elem.appendChild(superscript);
      }
    }
```

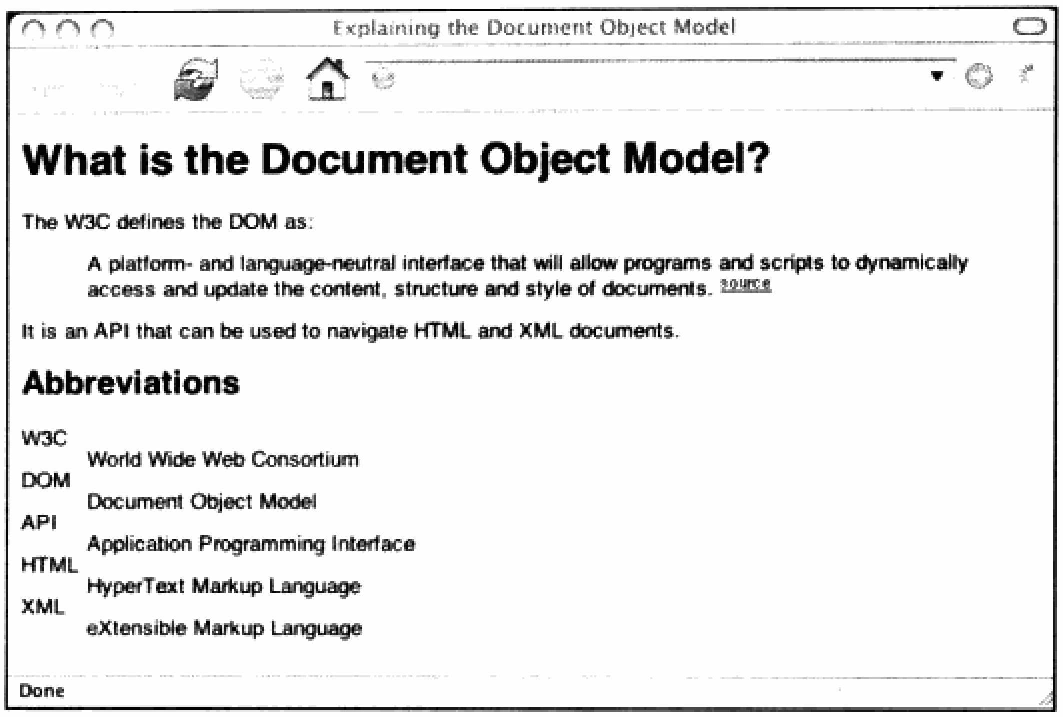
用 addLoadEvent()函数把 displayCitations()函数添加到 window.onload 事件处理函数上：

```
addLoadEvent(displayCitations);
```

为了调用 displayCitations.js 文件，我还需要在 explanation.html 文件的<head>部分添加一组<script>标签：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <meta http-equiv="content-type"
➥content="text/html; charset=utf-8" />
    <title>Explaining the Document Object Model</title>
    <link rel="stylesheet" type="text/css" media="screen"
➥href="styles/typography.css" />
    <script type="text/javascript" src="scripts/addLoadEvent.js">
    </script>
    <script type="text/javascript"
➥src="scripts/displayAbbreviations.js">
    </script>
    <script type="text/javascript" src="scripts/displayCitations.js">
    </script>
  </head>
<body>
    <h1>What is the Document Object Model?</h1>
    <p>
The <abbr title="World Wide Web Consortium">W3C</abbr> defines
➥the <abbr title="Document Object Model">DOM</abbr> as:
    </p>
    <blockquote cite="http://www.w3.org/DOM/">
      <p>
A platform- and language-neutral interface that will allow programs
➥and scripts to dynamically access and update the
➥content, structure and style of documents.
      </p>
    </blockquote>
    <p>
It is an <abbr title="Application Programming Interface">API</abbr>
➥that can be used to navigate <abbr title="HyperText Markup Language">
➥HTML</abbr> and <abbr title="eXtensible Markup Language">XML
➥</abbr> documents.
    </p>
  </body>
</html>
```

现在，把 explanation.html 文件加载到一个 Web 浏览器里就可以看到 displayCitations()函数的效果了。



8.6.3 显示“快速访问键清单”

我们此前编写的 displayAbbreviations()和 displayCitations()函数有许多共同之处：从创建一个由特定元素（abbr 元素或 blockquote 元素）构成的节点集合开始，用一个循环去遍历这个节点集合并在每次循环里创建出一些 HTML 内容，最后把新创建的 HTML 内容插入到文档里。

我将再向你们展示一个沿用了这一思路的例子。编写一个能够把文档里用到的所有快速访问键显示在页面里的函数。

1. HTML内容

accesskey 属性可以把一个元素（例如：一个链接）与键盘上的某个特定按键关联在一起。这对那些不能或不喜欢使用鼠标来浏览网页的人们很有用。如果你是一位有视力障碍的人士，键盘快捷方式肯定会给你带来许多方便。

一般来说，在适用于 Windows 系统的浏览器里，快速访问键的用法是在键盘上同时按下 Alt 键和特定按键；在适用于 Mac 系统的浏览器里，快速访问键的用法是同时按下 Ctrl 键和特定按键。

下面是 accesskey 属性是一个例子：

```
<a href="index.html" accesskey="1">Home</a>
```

请注意，设置太多的快速访问键往往会适得其反——它们或许会与浏览器内建的键盘快捷方式发生冲突。

如何设置快速访问键并无一定之规，但一些比较基本的快速访问键都有约定俗成的设置办法，对此感兴趣的读者可以浏览 http://www.clagnut.com/blog/193/。一般来说，accesskey="1"对应着一个“返回到本网站主页”的链接；accesskey="2"对应着一个“后退到前一页面”的链

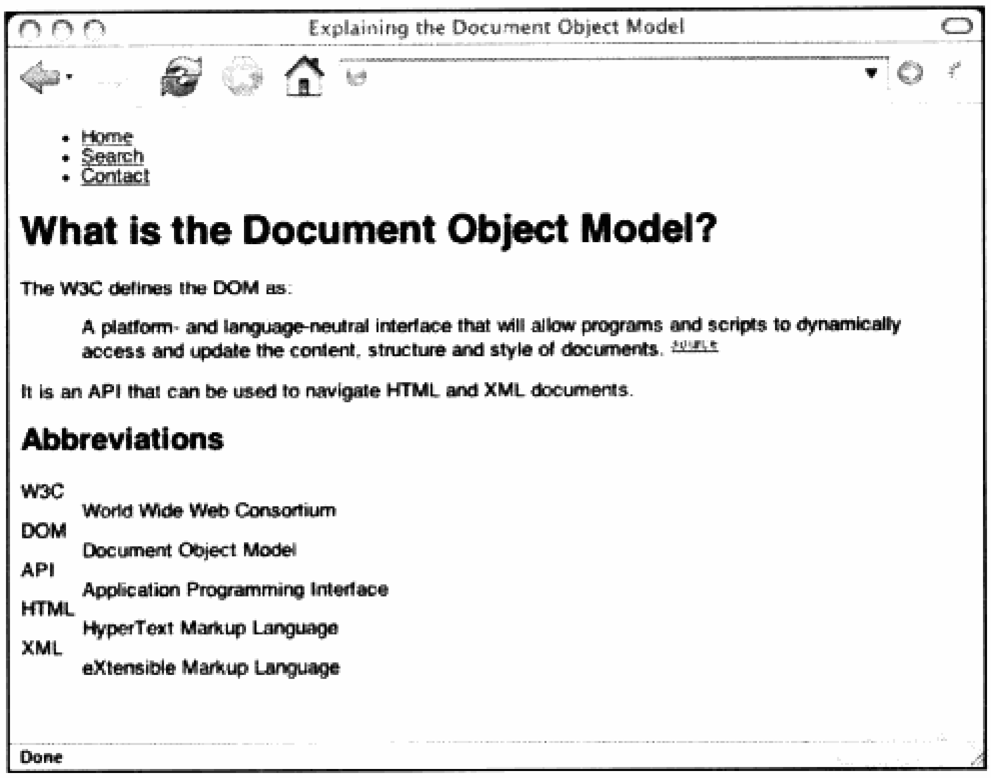
接；accesskey="4"对应着一个“打开本网站的搜索表单/页面”的链接；accesskey="9"对应着一个“本网站联系办法”的链接；accesskey="0"对应着一个“查看本网站的快速访问键清单”的链接。

以下 HTML 代码将生成一份快速访问键清单，这份清单是我们将在本小节研究的例子：

```
<ul id="navigation">
  <li><a href="index.html" accesskey="1">Home</a></li>
  <li><a href="search.html" accesskey="4">Search</a></li>
  <li><a href="contact.html" accesskey="9">Contact</a></li>
</ul>
```

请把这段 HTML 内容添加到 explanation.html 文档的<body>标签的开头。

现在，如果把 explanation.html 文档加载到一个浏览器里，你们就会看到这份清单里的链接，但看不到任何可以表明那些链接都有 accesskey 属性的迹象。



支持 accesskey 属性的浏览器有很多，但是否以及如何把快速访问键的分配情况显示在页面上却需要由身为网页设计人员的你们来决定。有许多网站都会在一个快速访问键清单（accessibility statement）页面上列明该网站都支持哪些快速访问键。

利用 DOM 技术，我们完全可以动态地创建一份快速访问键清单。下面是具体的步骤：

(1) 把文档里的所有链接全部提取到一个节点集合里。

(2) 遍历这个节点集合里的所有链接。

(3) 如果某个链接带有 acesskey 属性，就把它的值保存起来。

(4) 把这个链接在浏览器窗口里的屏显标识文字也保存起来。

(5) 创建一个清单（ul 元素），这个清单就是“快速访问键清单”。

(6) 为拥有快速访问键的各个链接分别创建一个列表项（li 元素）。
(7) 把列表项添加到"快速访问键清单"里。
(8) 把"快速访问键清单"添加到文档里。

下面是我按照以上步骤编写出来的 JavaScript 脚本代码。

2. JavaScript代码

我将把这个函数命名为 displayAccessKeys 并存入 displayAccessKeys.js 文件。

这个函数的工作原理与 displayAbbreviations()函数很相似：先把 accesskey 属性值和相关链接的屏显标识文本提取出来并存入一个关联数组，然后用一个 for/in 循环来遍历这个数组以创建"快速访问键清单"中的各个列表项。

下面就是我编写的 displayAccessKeys()函数的代码清单，因为未用到新技巧，我就不再对各条语句做详细说明了。代码清单里的注释语句应该可以把各个步骤解释清楚。

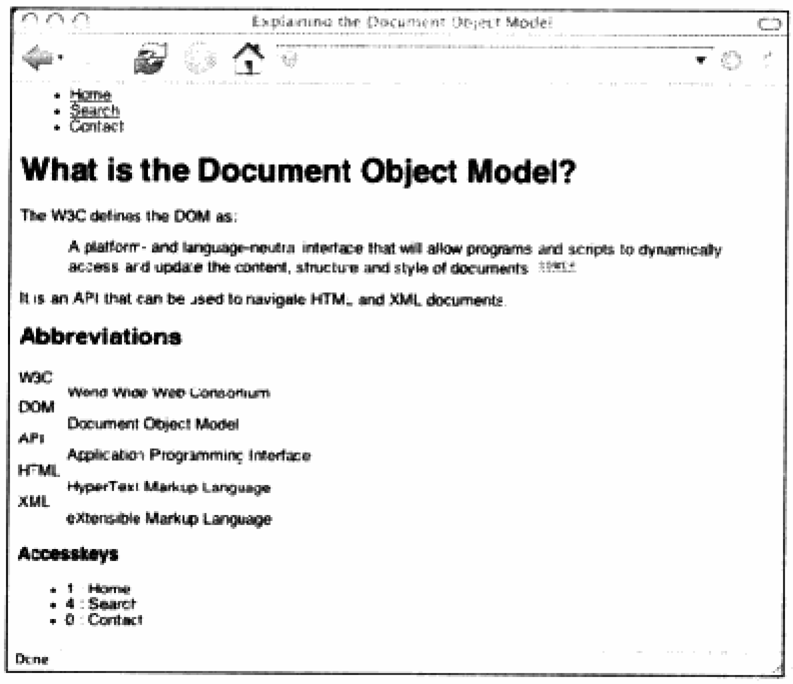
```
function displayAccesskeys() {
  if (!document.getElementsByTagName || !document.createElement ||
➥!document.createTextNode) return false;
// get all the links in the document
  var links = document.getElementsByTagName("a");
// create an array to store the access keys
  var akeys = new Array();
// loop through the links
  for (var i=0; i<links.length; i++) {
    var current_link = links[i];
// if there is no accesskey attribute, continue the loop
    if (!current_link.getAttribute("accesskey")) continue;
// get the value of the accesskey
    var key = current_link.getAttribute("accesskey");
// get the value of the link text
    var text = current_link.lastChild.nodeValue;
// add them to the array
    akeys[key] = text;
  }
// create the list
  var list = document.createElement("ul");
// loop through the access keys
  for (key in akeys) {
    var text = akeys[key];
//  create the string to put in the list item
    var str = key + ": "+text;
// create the list item
    var item = document.createElement("li");
    var item_text = document.createTextNode(str);
    item.appendChild(item_text);
// add the list item to the list
```

```
    list.appendChild(item);
  }
// create a headline
  var header = document.createElement("h3");
  var header_text = document.createTextNode("Accesskeys");
  header.appendChild(header_text);
// add the headline to the body
  document.body.appendChild(header);
// add the list to the body
  document.body.appendChild(list);
}
addLoadEvent(displayAccesskeys);
```

为了调用 displayAccessKeys.js 文件，我还需要在 explanation.html 文件的<head>部分添加一组<script>标签：

```
<script type="text/javascript"
src="scripts/displayAccesskeys.js"></script>
```

现在，如果把 explanation.html 文档加载到一个浏览器里，就可以看到动态创建的“快速访问键清单”。



8.7 小结

在这一章里，我们一起编写了几个很有用的 DOM 脚本，你们可以把这几个脚本用到任何一个网页里。虽然它们的用途不一样，但基本思路是相同的：用 JavaScript 函数先把文档结构里的一些现有信息提取出来，再把那些信息以一种清晰和有意义的方式重新插入到文档里去。

这些函数可以让网页变得更有条理、更容易浏览：

- ❑ 把文档里的缩略词语显示为一个“缩略词语表”。

- 为文档里引用的每段文献节选生成一个“文献来源链接”。
- 把文档所支持的快速访问键显示为一份“快速访问键清单”。

你们可以根据具体情况对这些脚本做进一步改进。比如说，我们这里是把“文献来源链接”直接显示在每个 blockquote 元素的后面，而你们可以把这些链接集中放在文档末尾的一个清单里——就像脚注那样。再比如说，我们这里生成了一份“快速访问键清单”，而你们可以把各个快速访问键分别追加在相关链接的末尾。

当然，还可以利用本章介绍的技巧去编写一些全新的脚本。例如，你们可以为文档生成一份目录：把文档里的 h1 和 h2 元素提取出来放入一份清单，再将其插入到文档的开头。你们甚至可以把这份清单里的列表项生成为一些可以让用户快速到达各有关标题的内部链接。

诸如此类的脚本还有很多，但在编写这类脚本时需要用到的DOM方法和属性却总是那几个。如果你们想通过 DOM 脚本去充实网页的内容，先制作一份结构良好的(X)HTML 文档将是最重要的前提条件。

在需要对文档里的现有信息进行检索时，以下 DOM 方法最有用：

- getElementById()
- getElementByTagName()
- getAttribute()

在需要把信息添加到文档里去时，以下 DOM 方法最有用：

- createElement()
- createTextNode()
- appendChild()
- insertBefore()
- setAttribute()

以上 DOM 方法的组合可以让我们编写出功能非常强大的 DOM 脚本来。

最后，我希望大家能够记住这样一点：JavaScript 脚本只应该用来充实文档的内容，要避免使用 DOM 技术来直接插入核心内容。

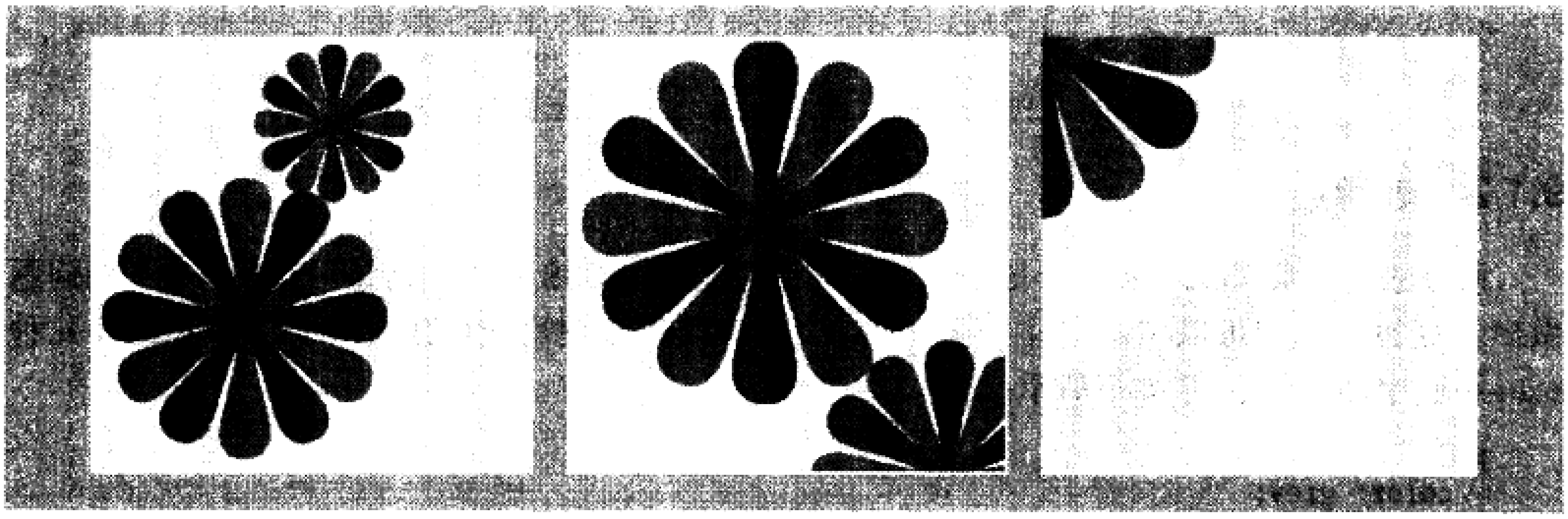
8.8　下章简介

至此，我们一直是在使用 JavaScript 语言和 DOM 技术去创建 HTML 内容。在下一章里，你们将看到 DOM 技术的一种全新应用：我将向大家展示如何运用 DOM 技术去处理诸如颜色、字体之类的文档样式信息。

DOM 技术不仅可以用来改变网页的结构，还可以用来对 HTML 页面元素的 CSS 信息进行刷新。

第 9 章

CSS-DOM



本章内容

- style 属性
- 如何检索样式信息
- 如何改变样式

在这一章里，Web 文档的表示层和行为层将正面接触。我将向大家展示如何利用 DOM 技术去检索（读）和设置（写）CSS 信息。

9.1 三位一体的网页

我们在浏览器里看到网页其实是由以下三层信息构成的一个共同体：

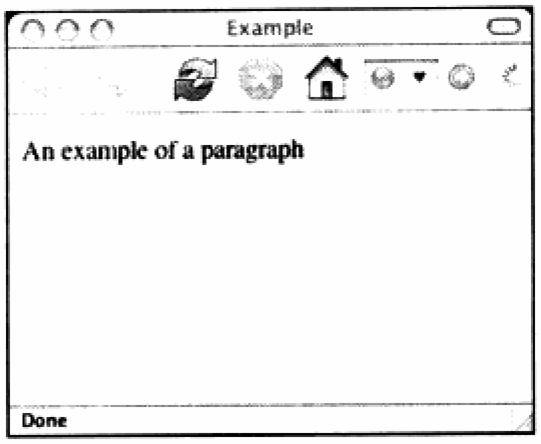
- 结构层
- 表示层
- 行为层

9.1.1 结构层

网页的结构层（structural layer）由 HTML 或 XHTML 之类的标记语言负责创建。标签（tag），也就是那些出现在尖括号里的单词，对网页内容的语义含义做出了描述，但这些标签不包含任何关于如何显示有关内容的信息。例如，<p>标签表达了这样一种语义："这是一个文本段。"如下

所示：

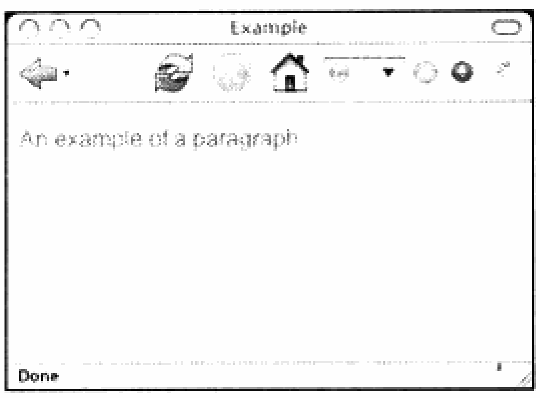
```
<p>An example of a paragraph</p>
```



9.1.2 表示层

网页的表示层（presentation layer）由 CSS 负责创建。CSS 对“如何显示有关内容”的问题做出了回答。我们可以定义这样一个 CSS：“文本段应该使用灰色的 Arial 字体和另外几种 scan-serif 字体来显示。”如下所示：

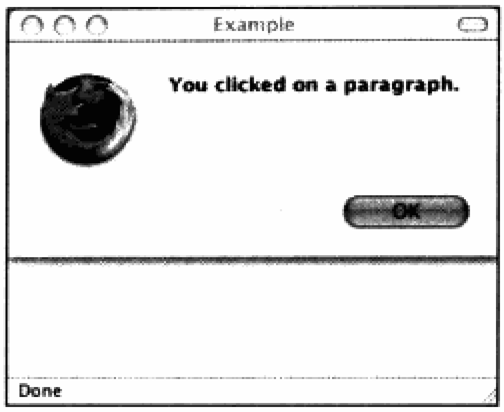
```
p {
  color: grey;
  font-family: "Arial", sans-serif;
}
```



9.1.3 行为层

网页的行为层（behavior layer）负责回答“内容应该如何对事件做出反应”这一问题。这是 JavaScript 语言和 DOM 主宰的领域。例如，我们可以利用 DOM 实现这样一种行为：“当用户点击一个文本段时，显示一个 alert 对话框。”如下所示：

```
var paras = document.getElementsByTagName("p");
for (var i=0; i<paras.length; i++) {
  paras[i].onclick = function() {
    alert("You clicked on a paragraph.");
  }
}
```



网页的表示层和行为层总是存在的，即使我们未明确地给出任何具体的指令也是如此。此时，Web 浏览器将把它的默认样式和默认事件处理函数施加在网页的结构层上。例如，浏览器会在呈现"文本段"元素时留出页边距，有些浏览器会在用户把鼠标指针悬停在某个元素的上方时弹出一个显示着该元素的 title 属性值的提示框，等等。

9.1.4 分离

在所有的产品设计活动中，选择最适用的工具去解决问题是最基本的原则。具体到网页设计工作，这意味着：

- 使用(X)HTML 去搭建文档的结构。
- 使用 CCS 去设置文档的呈现效果。
- 使用 DOM 脚本去实现文档的行为。

不过，在这三种技术之间存在着一些潜在的重叠区域，而我们也已见识过这样的例子：DOM 技术可以用来改变网页的结构，诸如 createElement()和 appendChild()之类的 DOM 方法能够动态地创建和添加 HTML 内容。

在 CSS 身上也可以找到这种技术相互重叠的例子。诸如:hover 和:focus 之类的预定义符号（术语称之为"伪 class 属性"）使我们可以根据用户触发事件来改变的呈现效果。改变元素的呈现效果当然是表示层的"势力范围"，但对用户触发事件做出反应却是行为层的领地。表示层和行为层的这种重叠形成了一个灰色地带。

伪 class 属性是 CSS 正在深入 DOM 领地的证据，但 DOM 在这方面也不是毫无作为。我们完全可以利用 DOM 技术把样式信息施加在 HTML 元素身上。

9.2 style 属性

文档中的每个元素都是一个对象。这些对象中的每一个又有着各种各样的属性。有些属性包含着某特定元素在节点树上的位置信息。比如说，parentNode、netxtSibling、previousSibling、ChildNodes、firstChild 和 lastChild 等属性所提供的信息都与节点在文档中的相对位置有关。

其他一些属性（比如 nodeType 和 nodeName 属性）包含着关于元素本身的信息。比如说，对

某个元素的 nodeName 属性进行的查询将返回一个诸如“p”之类的字符串。

除此之外，文档的每个元素节点还都有一个属性 style。style 属性包含着元素的样式信息，查询这个属性将返回一个对象而不是一个简单的字符串。样式信息都存放在这个对象的属性里：

```
element.style.property
```

下面是一个内嵌着样式信息的文本段元素的例子：

```
<p id="example" style="color: grey; font-family: 'Arial',sans-serif;">
An example of a paragraph
</p>
```

我们完全可以利用 style 属性把这个文本段的样式信息检索出来。

首先，需要从文档里把这个元素找出来。我已经给这个文本段设置了一个独一无二的 id 值 example。只需把这个 id 值传递给 getElementById()方法，再把这个方法的返回值赋值给变量 para，就可以通过 para 变量引用这个 p 元素了：

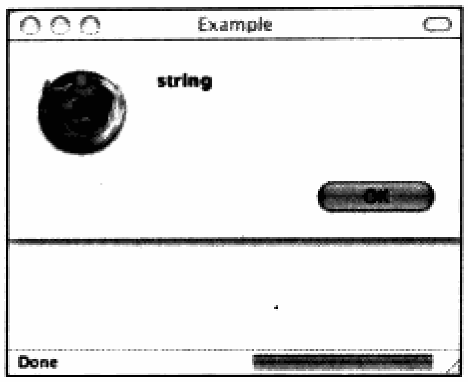
```
var para = document.getElementById("example");
```

开始检索这个元素的样式信息之前，我想先向大家证明一下 style 属性确实是一个对象。这一结论可以利用 JavaScript 关键字 typeof 来验证。下面，我们来对比一下 style 属性与 nodeName 属性的 typeof 结果。

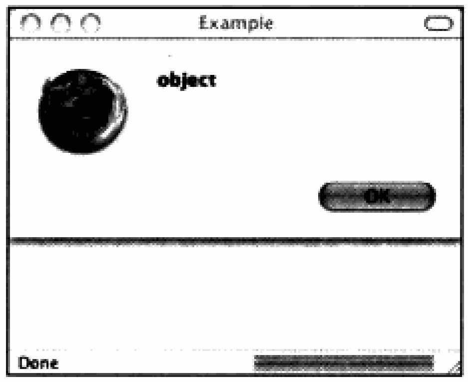
请先把下面这些 XHTML 代码写入文件 example.html，然后把这个文件加载到浏览器里：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Example</title>
  <script type="text/javascript">
window.onload = function() {
  var para = document.getElementById("example");
  alert(typeof para.nodeName);
  alert(typeof para.style);
}
  </script>
</head>
<body>
  <p id="example" style="color: grey; font-family:
➥ 'Arial',sans-serif;">
An example of a paragraph
  </p>
</body>
</html>
```

第一条 alert 语句将返回字符串“string”，这表明 nodeName 属性是一个字符串。



第二条 alert 语句将返回字符串“object”，这表明 style 属性是一个对象：



也就是说，不仅文档里的每个元素都是一个对象，它们的 style 属性也是一个对象。

9.2.1 样式信息的检索

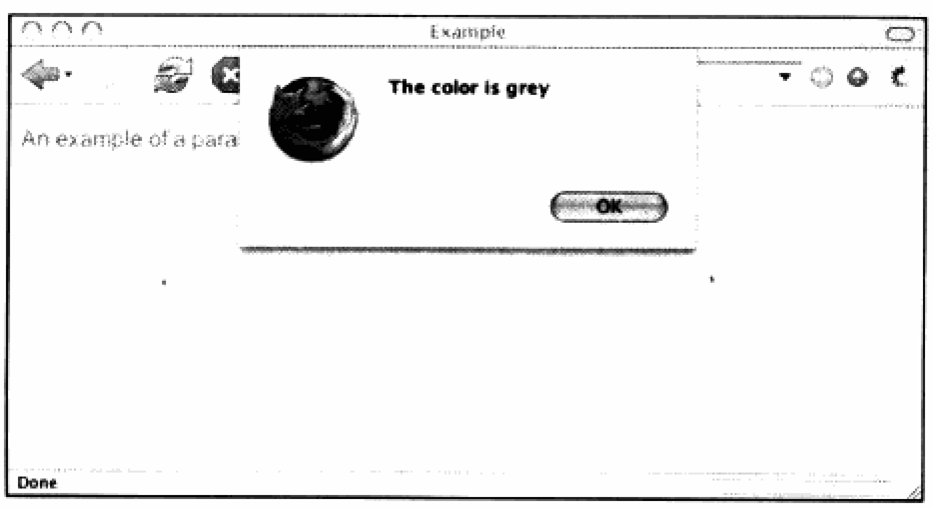
我们来看看如何检索 para 变量所代表的文本段的样式信息。为了查出某个元素在浏览器里的显示颜色，我们需要使用 style 对象的 color 属性：

element.style.color

下面这条 alert 语句将使我们看到 style 对象，（这个对象本身又是 para 元素的一个属性）的 color 属性：

```
alert("The color is " + para.style.color);
```

这个元素的 style 属性的 color 属性是“grey”。



我刚才还设置了 para 元素的 CSS 样式属性 font-family。这个属性的检索办法与 color 属性略有不同。我们不能简单地直接对 style 对象的 font-family 属性进行查询，因为“font”和“family”之间的连字符与 JavaScript 语言中的减法操作符相同，JavaScript 会把它解释为减号。如果像下面这样去检索一个名为 font-family 的属性，就会收到一条出错信息：

```
element.style.font-family
```

JavaScript 将把减号前面的内容解释为“*element* 元素的 style 属性的 font 属性”，把减号后面的内容解释为“一个名为 family 的变量”，把整个表达式解释为一个减法运算。所有这些完全违背了我的本意！

减号和加号之类的操作符是 JavaScript 语言保留给自己使用的特殊字符，不允许用在函数或变量的名字里。这同时意味着它们也不能用在方法或属性的名字里（别忘了，方法和属性其实是关联在某个对象上的函数和变量）。

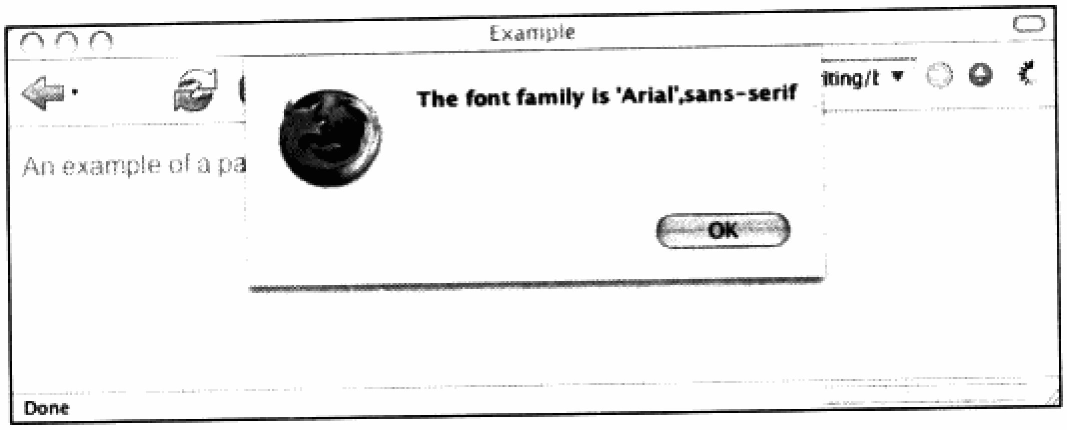
DOM 解决这个问题的办法是，采用“Camel 记号”来表示那些名字里有多个单词的 CSS 属性。也就是说，CSS 属性 font-family 在 DOM 脚本代码里应该写成 fontFamily：

```
element.style.fontFamily
```

为了查看 para 元素的 style 属性的 fontFamily 属性，我在 example.html 文件里增加了一条 alert 语句：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Example</title>
  <script type="text/javascript">
window.onload = function() {
  var para = document.getElementById("example");
  alert("The font family is " + para.style.fontFamily);
}
  </script>
</head>
<body>
  <p id="example" style="color: grey;
➥ font-family: 'Arial',sans-serif;">
An example of a paragraph
  </p>
</body>
</html>
```

在浏览器里重新加载 example.html 文件。



DOM 属性 fontFamily 的值与 CSS 属性 font-family 的值是一样。具体到这个例子，它是：

```
'Arial',sans-serif
```

不管 CSS 样式属性的名字里有多少个连字符，DOM 一律采用“Camel 记号”来表示它们。于是，CSS 属性 background-color 将对应于 DOM 属性 backgroundColor，CSS 属性 font-weight 将对应于 DOM 属性 fontWeight，DOM 属性 marginTopWidth 将对应于 CSS 属性 margin-top-width，等等。

有时，DOM 对样式属性的检索结果与它们的 CSS 设置值会采用不同的计量单位。

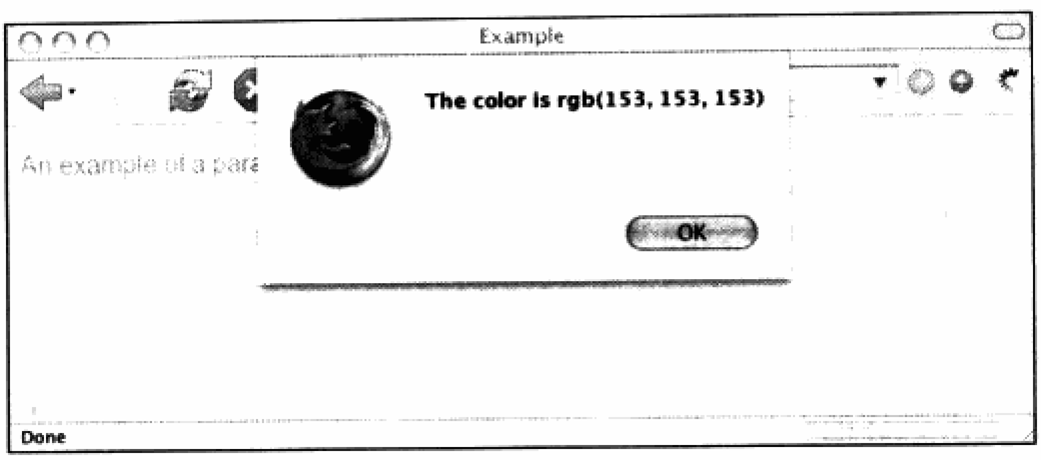
在示例文本段元素里，CSS 属性 color 的设置值是单词“grey”，用 JavaScript 代码检索出来的 DOM color 属性的值也是“grey”。现在，如果我把那个文本段里的 color 属性值改为十六进制值#999999：

```
<p id="example" style="color: #999999; font-family: 'Arial',sans-serif">
```

再在 JavaScript 代码里用一条 alert 语句去输出 DOM 上下文里的 color 属性：

```
alert("The color is " + para.style.color);
```

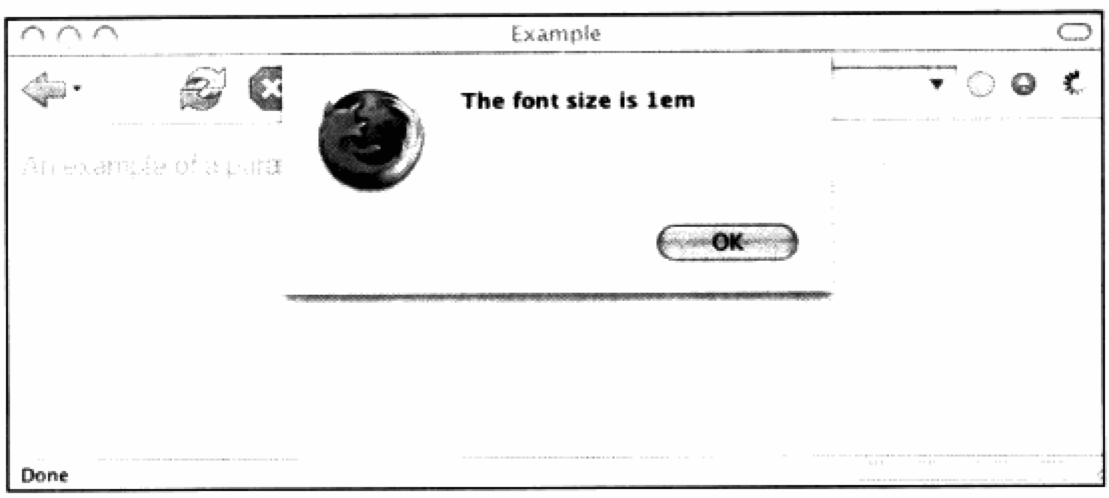
那么，在某些浏览器里，color 属性的返回值将是一个 RGB（红，绿，蓝）颜色值(153,153,153)，如下所示。



还好，这类例外情况并不多。绝大部分样式属性的返回值与它们的设置值都采用同样的计量单位。如果我们在设置 CSS font-size 属性时以 em 为单位，相应的 DOM fontSize 属性也将以 em 为单位：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Example</title>
  <script type="text/javascript">
window.onload = function() {
  var para = document.getElementById("example");
  alert("The font size is " + para.style.fontSize);
}
  </script>
</head>
<body>
  <p id="example" style="color: grey; font-family: 'Arial',sans-serif;
➥ font-size: 1em;">
An example of a paragraph
  </p>
</body>
</html>
```

如下图所示，DOM fontSize 属性的确也是以 em 为单位的。



如果 CSS font-size 属性的值是 1em，DOM fontSize 属性的返回值就将是 1em。如果 CSS font-size 属性的值是 12px，DOM fontSize 属性的返回值就将是 12px。

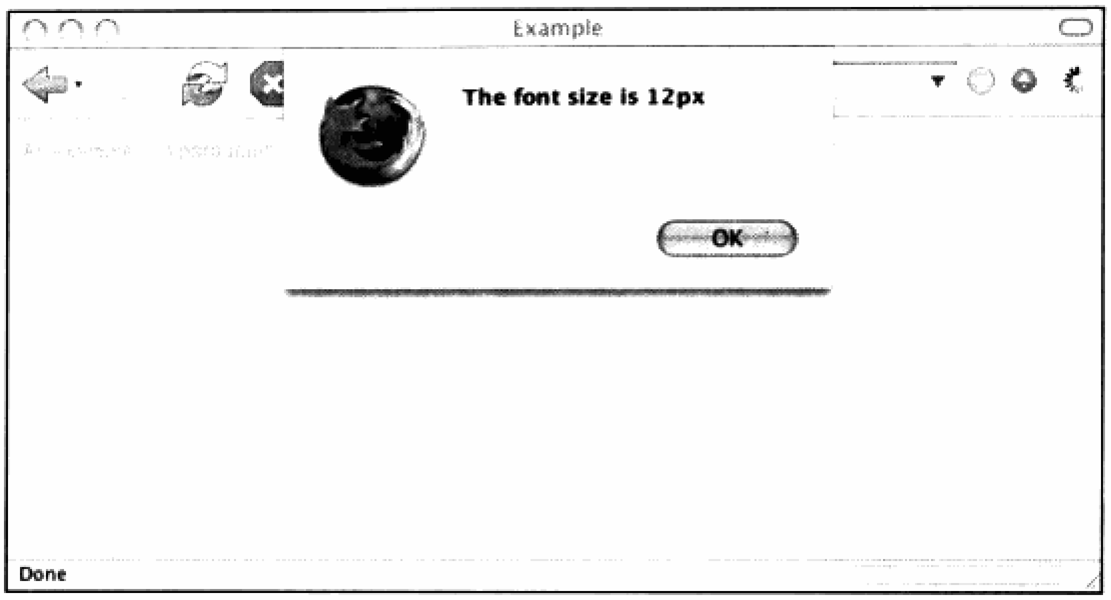
CSS 还提供了一些能够一次声明多个样式的多用途属性。比如说，如果你声明了 font: 12px 'Arial'，sans-serif，CSS font-size 属性将被设置为 12px，CSS font-family 属性将被设置为 'Arial'，sans-serif：

```
<p id="example" style="color: grey; font: 12px 'Arial',sans-serif;">
```

DOM 能够理解像 font 这样的多用途属性。如果查询 fontSize 属性，将得到一个 12px 的值：

```
alert("The font size is " + para.style.fontSize);
```

如下图所示，DOM fontSize 属性的确也是以 px 为单位的。



内嵌样式

通过 style 属性检索样式信息有个很大的局限性。

style 属性只能返回那些内嵌在 HTML 内容里的样式信息。换句话说，只有把 CSS style 属性插入到 HTML 代码里，才可以用 DOM style 属性去查询那些信息：

```
<p id="example" style="color: grey; font: 12px 'Arial',sans-serif;">
```

这可不是使用样式的最佳办法——文档的表示层与结构层混杂在一起了。更好的办法是用一个下面这样的外部样式表去设置各有关样式：

```
p#example {
  color: grey;
  font: 12px 'Arial', sans-serif;
}
```

请把上面这段 CSS 代码存入文件 styles.css。然后，从 example.html 文件里把内嵌在 HTML 代码里的样式信息删掉，只保留以下内容：

```
<p id="example">
An example of a paragraph
</p>
```

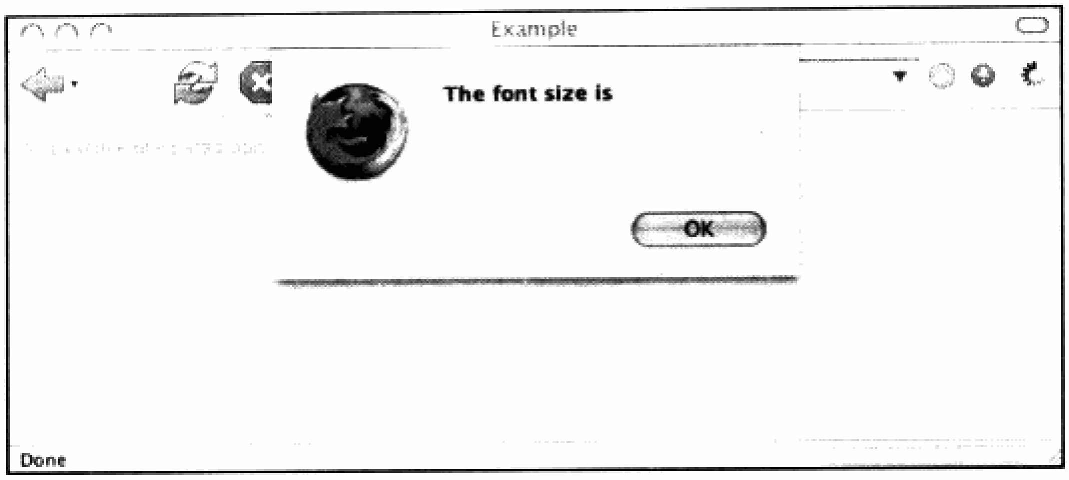
在 example.html 文件的开头部分加上一个 link 元素并让它指向 styles.css 文件：

```
<link rel="stylesheet" type="text/css" media="screen"
➥ href="styles.css" />
```

这次，样式信息还像以前那样作用到了 HTML 内容上，但与使用 style 属性时的情况不同，来自外部文件 styles.css 的样式信息已经不能再用 DOM style 属性检索出来了：

```
alert("The font size is " + para.style.fontSize);
```

DOM style 属性不能用来检索在外部 CSS 文件里声明的样式信息。



如果把样式信息添加在 example.html 文件<head>部分的<style>标签里，你将看到类似的结果：

```
<style type="text/css">
  p#example {
    color: grey;
    font: 12px 'Arial', sans-serif;
  }
</style>
```

DOM style 属性提取不到如此设置的样式信息。

在外部样式表里声明的样式信息不会进入 style 对象，在文档的<head>部分里声明的样式信息也是如此。

style 对象只包含着在 HTML 代码里用 style 属性声明的样式信息。但这几乎没有实用价值，因为样式信息应该与 HTML 内容分离开来。

现在，你们或许会认为用 DOM 去处理 CSS 样式信息毫无意义，但这里还有另一种情况可以让 DOM style 对象能够正确地反映出我们设置的样式信息：如果用 DOM 来设置样式信息，就可以用 DOM 再把它们检索出来。

9.2.2　设置样式信息

许多 DOM 属性是只读的——我们只能用它们来检索信息，不能用它们来设置或刷新信息。如果你想收集关于某个元素在节点树里的位置信息，诸如 previousSibling、netxtSibling、parentNode、firstChild 和 lastChild 之类的属性可以帮上你的大忙，但它们都不能用来设置或刷新有关的信息。

凡事无绝对，style 对象的各个属性就都是可读写的。我们不仅可以通过某个元素的 style 属性去检索信息，还可以通过它去设置和刷新信息。我们可以用赋值操作符——也就是等号来做到这一点：

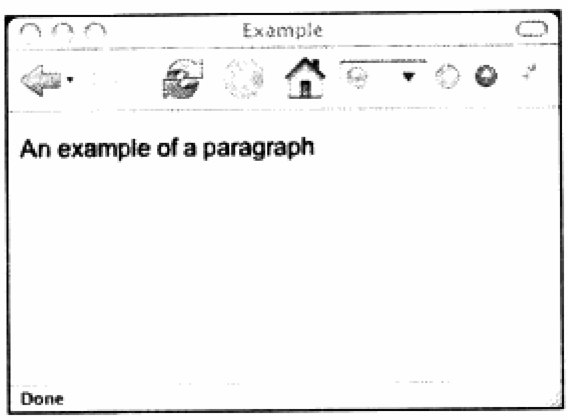
```
element.style.property = value
```

style 对象的各个属性的值永远是一个字符串。现在来验证一下：在 example.html 文件里用一

些 JavaScript 代码覆盖那些内嵌在 HTML 内容里的 CSS 代码——比如说，把 para 元素的 color 属性设置为"black"：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Example</title>
  <script type="text/javascript">
window.onload = function() {
  var para = document.getElementById("example");
  para.style.color = "black";
}
  </script>
</head>
<body>
  <p id="example" style="color: grey;
➥ font-family: 'Arial',sans-serif;">
An example of a paragraph
  </p>
</body>
</html>
```

color 属性已经被刷新为 black 了，如图所示。



请注意，style 对象的属性值必须放在引号里，单引号或双引号均可：

```
para.style.color = 'black';
```

如果忘了使用引号，JavaScript 将把等号右边的值解释为一个变量：

```
para.style.color = black;
```

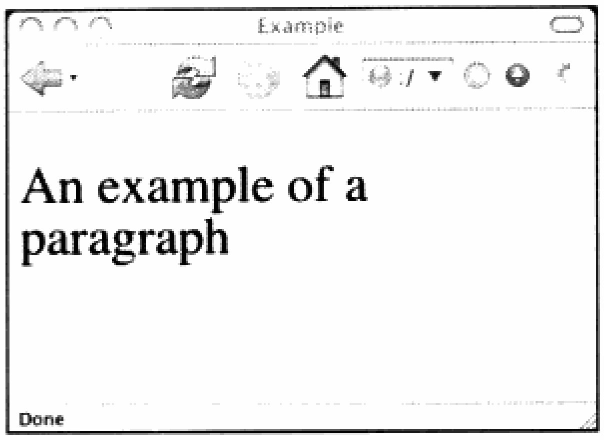
如果你此前未声明过变量 black，上面这行代码将无法工作。

赋值操作符可以用来对任何一种样式属性——诸如 font 之类的多用途属性也不例外进行设置：

```
para.style.font = "2em 'Times',serif";
```

上面这条语句将把 fontSize 属性设置为 2em，把 fontFamily 属性设置为'Times'，serif，如右

图所示。



通过 JavaScript 代码设置样式信息并不难，在上面我也给出了一些具体的例子。但在决定这么做之前，应该先弄清楚：我为什么要这么做？

9.3 何时该用 DOM 脚本去设置样式信息

你们已经见识过用 DOM 脚本代码设置样式信息是多么的容易了，但你能够做什么并不意味着你应该那么做。在绝大多数场合，还是应该使用 CSS 去声明样式信息。就像你不应该利用 DOM 技术去创建重要的 HTML 内容那样，你也不应该利用 DOM 技术去为文档设置重要的样式信息。

不过，有时可以利用 DOM 脚本对文档的样式做进一步的充实，但那很不明智。

9.3.1 根据元素在节点树里的位置来设置样式信息

通过 CSS 声明样式信息的具体做法主要有三种。第一种是为同类型的所有元素（比如 p 元素）统一地声明一种样式，如下所示：

```
p {
  font-size: 1em;
}
```

第二种是为有着特定 class 属性的所有元素统一地声明一种样式，如下所示：

```
.fineprint {
  font-size: .8em;
}
```

第三种是为某个有着独一无二的 id 属性的元素单独声明一种样式，如下所示：

```
#intro {
  font-size: 1.2em;
}
```

不过，至少在目前，我们还无法使用 CSS 根据某个元素在节点树里的位置来为它声明一种样式。例如，目前还不能使用 CSS 做到“把以下样式施加在所有 h1 元素的下一个兄弟节点上。”

现在，CSS 还无法根据元素之间的相对位置关系找出某个特定的元素来，但这对 DOM 来说却不是什么难题。我们可以利用 DOM 代码轻而易举地找出文档中的所有 h1 元素，然后再同样轻而易举地找出紧跟在每个 h1 元素后面的那个元素并把样式信息添加给它。

首先，用 getElementByTagName()方法把所有的 h1 元素找出来：

```
var headers = document.getElementsByTagName("h1");
```

然后，遍历这个节点集合里所有元素：

```
for (var i=0; i<headers.length; i++) {
```

文档中的下一个节点可以用 nextSibling 属性查找出来：

```
headers[i].nextSibling
```

请注意，我们这里真正需要的不是“下一个节点”，而是“下一个元素节点”。下面这个 getNextElement()函数可以让我们轻松完成这一任务：

```
function getNextElement(node) {
  if(node.nodeType == 1) {
   return node;
  }
  if (node.nextSibling) {
    return getNextElement(node.nextSibling);
  }
  return null;
}
```

把当前 h1 元素（即 headers[i]）的 nextSibling 节点作为参数传递给 getNextElement()函数，并把这个函数调用的返回值赋值给 elem 变量：

```
var elem = getNextElement(headers[i].nextSibling);
```

现在，就可以按照我们的想法去设置这个元素的样式了：

```
elem.style.fontWeight = "bold";
elem.style.fontSize = "1.2em";
```

最后，把以上代码写入函数 styleHeaderSiblings——别忘了安排一些测试去检查浏览器能否理解我们在这个函数里用到的 DOM 方法：

```
function styleHeaderSiblings() {
  if (!document.getElementsByTagName) return false;
  var headers = document.getElementsByTagName("h1");
  for (var i=0; i<headers.length; i++) {
    var elem = getNextElement(headers[i].nextSibling);
    elem.style.fontWeight = "bold";
    elem.style.fontSize = "1.2em";
  }
}
```

```
function getNextElement(node) {
  if(node.nodeType == 1) {
    return node;
  }
  if (node.nextSibling) {
    return getNextElement(node.nextSibling);
  }
  return null;
}
```

你们可以用 window.onload 事件去调用这个函数：

```
window.onload = styleHeaderSiblings;
```

但为了让自己以后能把更多的函数方便地添加到 window.onload 事件上，我决定用 addLoadEvent()函数来完成这一任务：

```
addLoadEvent(styleHeaderSiblings);
```

下面是 addLoadEvent()函数的代码清单，我们可以把它保存到外部文件里：

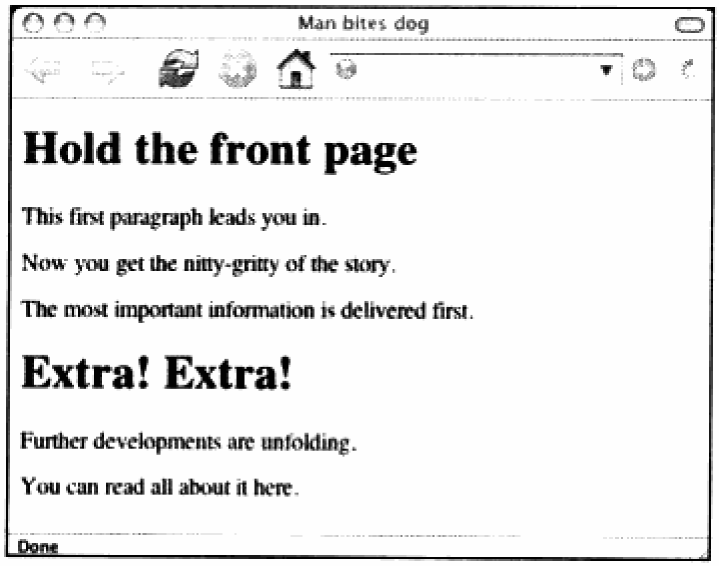
```
function addLoadEvent(func) {
  var oldonload = window.onload;
  if (typeof window.onload != 'function') {
    window.onload = func;
  } else {
    window.onload = function() {
      oldonload();
      func();
    }
  }
}
```

为了看到 styleHeaderSiblings()函数的使用效果，我还需要在 HTML 文档里添加一些一级标题（即 h1 元素）：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Man bites dog</title>
</head>
<body>
  <h1>Hold the front page</h1>
  <p>This first paragraph leads you in.</p>
  <p>Now you get the nitty-gritty of the story.</p>
  <p>The most important information is delivered first.</p>
  <h1>Extra! Extra!</h1>
  <p>Further developments are unfolding.</p>
  <p>You can read all about it here.</p>
```

```
</body>
</html>
```

把这个 HTML 文档保存为 story.html 文件；下面是它目前在浏览器里的样子。

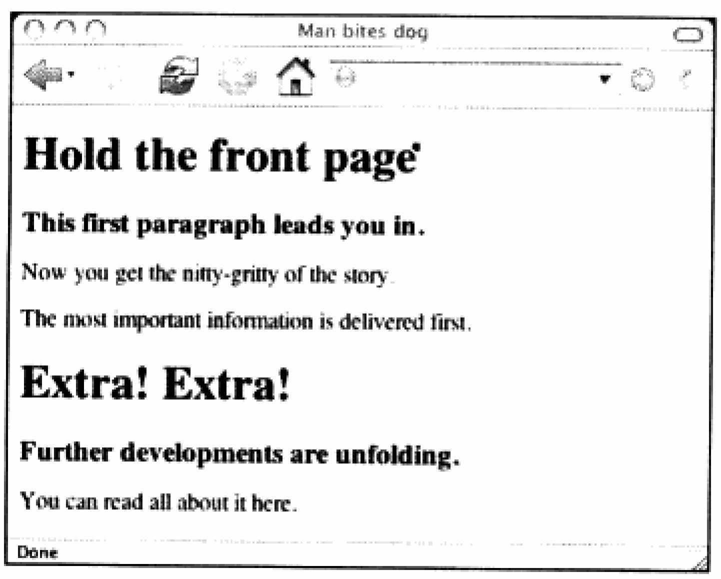


接下来，创建文件夹 scripts 来存放我的 JavaScript 脚本文件。我将把 addLoadEvent()函数写入一个名为 addLoadEvent.js 的文件并把它放到这个文件夹里，我还将把 styleHeaderSiblings()函数写入一个名为 styleHeaderSiblings.js 的文件并把它也放到此文件夹里。

为了调用这个两个 JavaScript 脚本文件，还需要在 story.html 文件的<head>部分插入一些<script>标签：

```
<script type="text/javascript" src="scripts/addLoadEvent.js">
</script>
<script type="text/javascript" src="scripts/styleHeaderSiblings.js">
</script>
```

现在，把 story.html 文件加载到 Web 浏览器里就可以看到由 DOM 脚本代码生成的样式的效果了。我们动态设置的样式将作用于紧跟在各个 h1 元素后面的那个元素。



从理论上讲，这类样式还是应该用 CSS 来设置；但在实践中，用 CSS 来设置这类样式的难度往往会很大。具体到这个例子，其实我们只需给紧跟在 h1 元素后面的每个元素添加一个 class 属性，就可以用 CSS 来获得同样的效果。但如果文档的内容需要定期编辑和刷新的话，添加 class 属性的工作很快就会变成一种负担。不仅如此，如果文档的内容需要通过一个 CMS（内容管理系统）来处理的话，给文档内容的特定部分添加 class 属性或其他样式信息的做法甚至可能是不允许的。

可喜的是，CSS2（第二代 CSS 标准）已经在这方面做了一些准备——CSS2 增加了一些诸如:first-child 和:last-child 之类的伪 class 属性。就目前而言，浏览器对这些伪 class 属性的支持还处于萌芽阶段，不是没有，就是还不够完善。在 CSS2 取代 CSS 成为主流技术之前，用 DOM 来弥补它们之间的技术鸿沟会是一个非常不错的解决方案。

9.3.2　根据某种条件来设置样式信息

这里不妨假设我有一份由一些日期和地名构成的清单，比如一份乐队演出日程表或一份旅行日程表。我们不必关心它到底是什么，只要其中的日期和地点有着直接对应的关系就行了。这类数据最适于制作成表格，而把表格数据转换为 HTML 内容的最佳工具当然非<table>标签莫属。

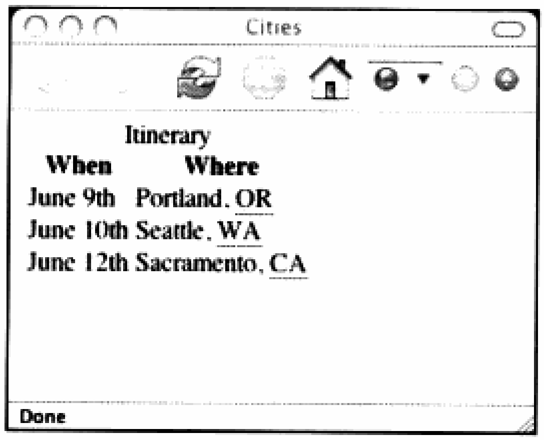
注意　*在用CSS布置和美化网页时，千万不要人云亦云地认为表格都是不好的。利用表格来安排页面元素的屏幕显示位置不一定是个好主意，利用表格来显示表格数据却是理所应当的。*

下面是为这个表格编写的 HTML 代码：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Cities</title>
</head>
<body>
  <table>
    <caption>Itinerary</caption>
    <thead>
    <tr>
      <th>When</th>
      <th>Where</th>
    </tr>
    </thead>
    <tbody>
    <tr>
      <td>June 9th</td>
      <td>Portland, <abbr title="Oregon">OR</abbr></td>
```

```
    </tr>
    <tr>
      <td>June 10th</td>
      <td>Seattle, <abbr title="Washington">WA</abbr></td>
    </tr>
    <tr>
      <td>June 12th</td>
      <td>Sacramento, <abbr title="California">CA</abbr></td>
    </tr>
    </tbody>
  </table>
</body>
</html>
```

把这些 HTML 代码保存为 itinerary.html 文件。如果现在就把这个文件加载到一个 Web 浏览器里，你们将看到一个包含着全部的信息但没有美感可言的表格。



为了使其变得整齐美观，我为它编写了一个如下所示的 CSS 样式表：

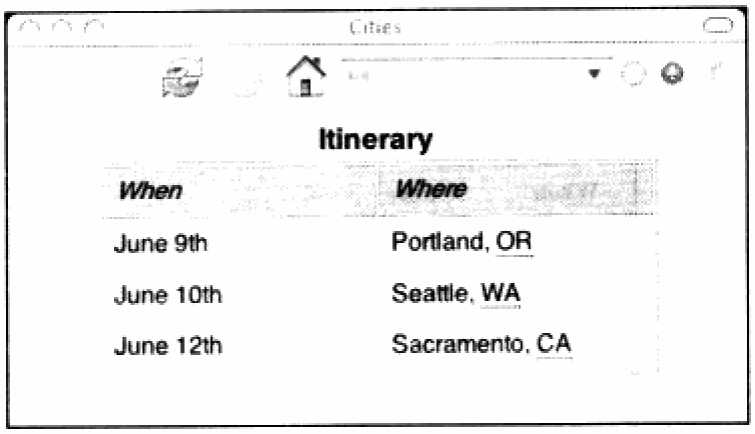
```
body {
  font-family: "Helvetica","Arial",sans-serif;
  background-color: #fff;
  color: #000;
}
table {
  margin: auto;
  border: 1px solid #699;
}
caption {
  margin: auto;
  padding: .2em;
  font-size: 1.2em;
  font-weight: bold;
}
th {
  font-weight: normal;
  font-style: italic;
  text-align: left;
```

```
  border: 1px dotted #699;
  background-color: #9cc;
  color: #000;
}
th,td {
  width: 10em;
  padding: .5em;
}
```

把这个 CSS 样式表保存为 format.css 文件并将其放入文件夹 styles 里。在 itinerary.html 文档的<head>部分增加一个<link>标签来引用这个 CSS 文件：

```
<link rel="stylesheet" type="text/css" media="screen"
➥ href="styles/format.css" />
```

现在，在 Web 浏览器里刷新 itinerary.html 文件就可以看到这个 CSS 的作用效果。



让表格里的各行数据更加清晰醒目的常用技巧之一是，交替改变它们的背景颜色，如此形成的斑马线效果会让两个相邻的表格行更加泾渭分明。这种效果可以通过“每隔一行就改变一次样式”的办法来获得。具体到 itinerary.html 文档这个例子，我们只需为表格中的每个奇数行（或每个偶数行）设置一个 class 属性即可。不过，这个办法非常不方便，尤其是对那些大表格而言——如果你以后需要在这个表格的半中间插入或删除表格行，你将不得不以手动方式去刷新大量的 class 属性。

对 CSS 来说，“在清单/表格里每隔一行就改变一次样式”可不容易。但这种重复性的工作对 JavaScript 来说却是小菜一碟。我们可以用一个 while 或 for 循环轻松地遍历一个很长的清单/表格。

我们可以编写一个函数来为表格“画上斑马线”，只要每隔一行设置一次样式信息就行了：

(1) 把文档里的所有 table 元素找出来。
(2) 对每个 table 元素，创建 odd 变量并把它初始化为 false。
(3) 遍历这个表格里的所有数据行。
(4) 如果变量 odd 的值是 true，设置样式信息并把 odd 变量修改为 false。
(5) 如果变量 odd 的值是 false，不设置样式信息，但把 odd 变量修改为 true。

我为这个函数命名为“stripeTables”。这个函数不需要参数，所以函数名后面的圆括号将是

空的。别忘了在这个函数的开头部分安排一些测试，以检查浏览器是否理解这个函数所用到的 DOM 方法：

```
function stripeTables() {
  if (!document.getElementsByTagName) return false;
  var tables = document.getElementsByTagName("table");
  for (var i=0; i<tables.length; i++) {
    var odd = false;
    var rows = tables[i].getElementsByTagName("tr");
    for (var j=0; j<rows.length; j++) {
      if (odd == true) {
        rows[j].style.backgroundColor = "#ffc";
        odd = false;
      } else {
        odd = true;
      }
    }
  }
}
```

这个函数应该在页面加载时得到执行。用 addLoadEvent()函数来安排这种事是最好的办法：

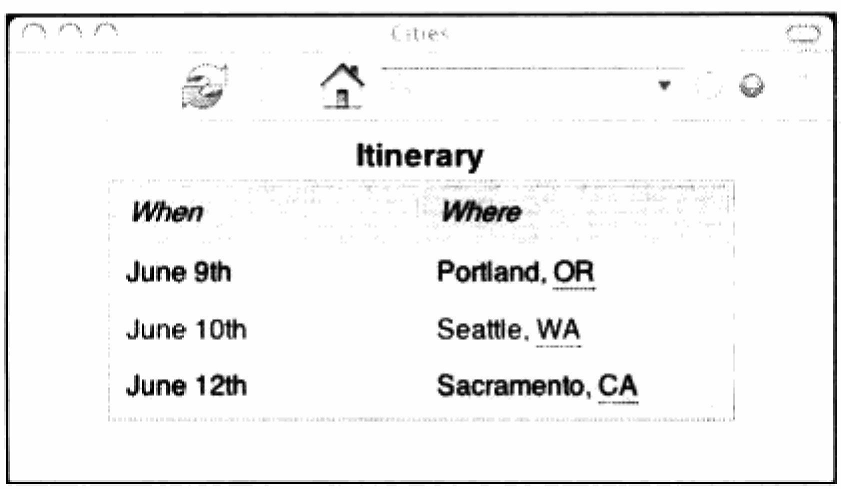
```
addLoadEvent(stripeTables);
```

把以上 JavaScript 代码保存为文件 stripeTables.js，再将其和 addLoadEvent.js 文件都放到文件夹 scripts 里去。

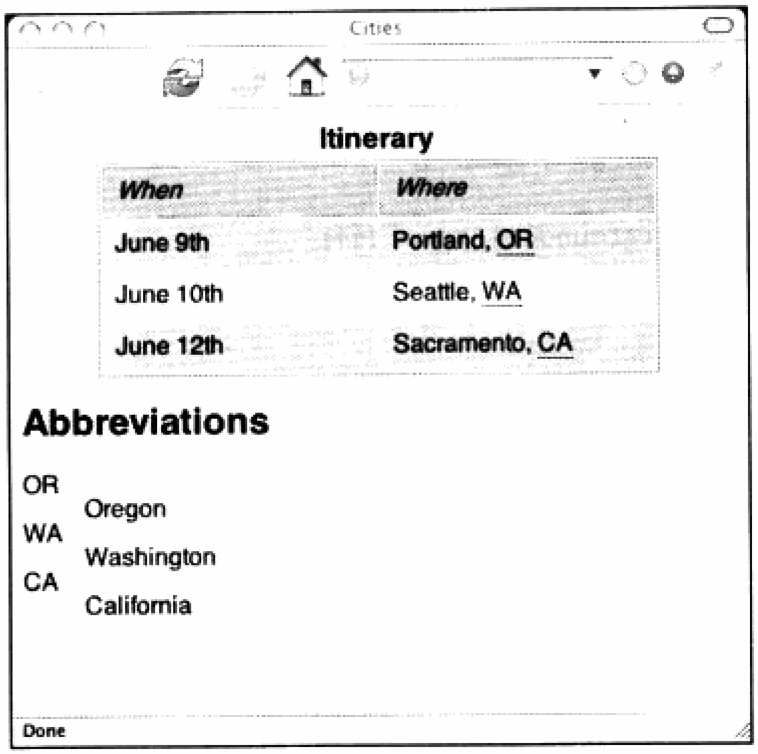
在 itinerary.html 文档的<head>部分，增加两个<script>标签来调用这两个 JavaScript 脚本文件：

```
<script type="text/javascript" src="scripts/addLoadEvent.js">
</script>
<script type="text/javascript" src="scripts/stripeTables.js">
</script>
```

现在，如果把 itinerary.html 文件加载到一个 Web 浏览器里，就可以看到表格里的偶数行都有了一个新的背景颜色。



很凑巧，我们还可以把来自上一章的 displayAbbreviations()函数用在这个文档上。把 displayAbbreviations.js 文件也放到 scripts 文件夹里，并在 itinerary.html 文档的<head>部分再增加一个<script>标签。在 Web 浏览器里刷新这个页面就可以看到由 JavaScript 代码动态生成的“缩略词语表”了。



9.3.3 对事件做出响应

只要有可能，就应该尽可能地选用 CSS 来为文档设置各种样式。可话虽如此，你们刚才也看到一些 CSS 不能或是难以满足应用要求的情况。在 CSS 力不从心的场合，DOM 可以帮上我们的大忙。

何时应该使用 CSS 来设置样式，何时应该使用 DOM 来设置样式并不总是那么容易判断。如果问题涉及需要根据某个事件来改变样式，这种判断就更难做出了。

CSS 提供的:hover 等伪 class 属性允许我们根据 HTML 元素的状态来改变样式。DOM 也可以通过 onmouseover 等事件处理函数对 HTML 元素的状态变化做出响应。何时应该使用:hover 属性、何时应该使用 onmouseover 事件处理函数是很难做出判断的。

最简单的解决方案是选择最容易实现的办法。比如说，如果你们只是想让链接在鼠标指针悬停在其上方时改变颜色，就应该选用 CSS 来解决这个问题：

```
a:hover {
  color: #c60;
}
```

伪 class 属性:hover 已经得到了绝大多数浏览器的支持——至少在它被用来改变链接的样式时是如此。但如果还想利用这个伪 class 属性在鼠标指针悬停在其他元素的上方时改变它们的样

式，支持这种用法的浏览器可就没有那么多了。

仍以 itinerary.html 文档中的表格为例。如果你想让某个表格行在鼠标指针悬停在它的上方时呈选中状态，你可以使用 CSS：

```
tr:hover {
  font-weight: bold;
}
```

从理论上讲，这应该让鼠标指针悬停在它上方的那个表格行的文本在浏览器画面里呈现出黑体字的显示效果；但在实践中，这种效果只能在一部分浏览器里看到。

在类似于这样的场合，DOM 可以确保你的想法能够得到实现。对绝大多数的现代浏览器而言，对 CSS 伪 class 属性的支持或许还有不足，但对 DOM 的支持却都没有问题。在浏览器们对 CSS 的支持功能进一步完善之前，如果你打算根据某些个事件去改变 HTML 元素的样式，用 DOM 来实现你的想法会更切合实际。

下面这个 highlightRows()函数将在鼠标指针悬停在某个表格行的上方时，把该行文本显示为黑体字：

```
function highlightRows() {
  if(!document.getElementsByTagName) return false;
  var rows = document.getElementsByTagName("tr");
  for (var i=0; i<rows.length; i++) {
    rows[i].onmouseover = function() {
      this.style.fontWeight = "bold";
    }
    rows[i].onmouseout = function() {
      this.style.fontWeight = "normal";
    }
  }
}
addLoadEvent(highlightRows);
```

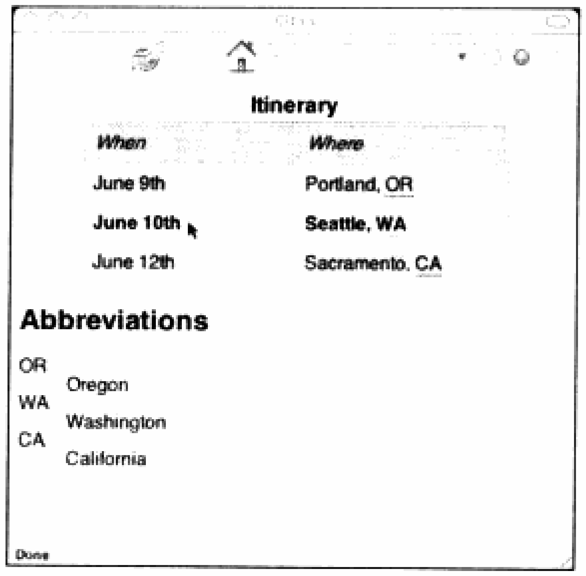
请把这个函数存入文件 highlightRows.js 并把它放入 scripts 文件夹，然后在 itinerary.html 文档的<head>部分增加一个如下所示的<script>标签：

```
<script type="text/javascript" src="scripts/highlightRows.js">
</script>
```

在 Web 浏览器里刷新 itinerary.html 文档。现在，当你把鼠标指针悬停在某个表格行的上方时，这个表格行里的文本将变成黑体字：

此时，我们将需要决定是使用纯粹的 CSS 解决方案，还是利用 DOM 来设置样式。我们将需要考虑以下因素：

- 这个问题最简单的解决方案是什么
- 哪种解决方案会得到更多浏览器的支持



要想做出明智的抉择，就必须对 CSS 和 DOM 脚本设计技术都有着足够深入的了解。有句老话说得好：如果你手里只有榔头，那么你看到的任何东西就都像是一颗钉子。如果你只喜欢使用 CSS，你十有八九会选择一个 CSS 解决方案，而不考虑 JavaScript 解决方案的效果会不会更好。反之，如果你只懂得编写 DOM 脚本，你往往会立刻动手编写 JavaScript 函数，而不可考虑用 CSS 来解决问题会不会更简明快捷。

这里的一般原则是：如果你想改变某个元素的呈现效果，就应该选用 CSS；如果你想改变某个元素的行为，就应该选用 DOM；如果你想根据某个元素的行为去改变它的呈现效果，请运用你的理智——在这个问题上没有放之四海而皆准的解决方案。

9.4　className 属性

在本章前面的例子里，我们一直是在使用 DOM 直接设置或修改样式信息。这种做法并不值得推荐，因为那是在让“行为层”去完成“表示层”的工作。如果你想改变 DOM 脚本设置的样式信息，你将不得不埋头于 JavaScript 函数去寻找和修改有关的语句。如果你可以在样式表里进行那些修改，事情就会容易得多，也合情合理得多。

这里有一种简明的解决方案：与其使用 DOM 脚本去直接改变某个元素的样式信息，不如通过 JavaScript 代码去刷新这个元素的 class 属性。

请大家仔细看看 styleHeaderSiblings()函数是如何添加样式信息的，并想想这有什么不好：

```
function styleHeaderSiblings() {
  if (!document.getElementsByTagName) return false;
  var headers = document.getElementsByTagName("h1");
  for (var i=0; i<headers.length; i++) {
    var elem = getNextElement(headers[i].nextSibling);
    elem.style.fontWeight = "bold";
    elem.style.fontSize = "1.2em";
  }
}
```

看出问题了吗？如果以后想把紧跟在一级标题之后的那个元素的 CSS 字号值从 1.2em 改为 1.4em，我们将不得不去修改 styleHeaderSiblings()函数。

如果 CSS 样式表里有一条如下所示的样式声明（对应于 class="intro"），事情会简单许多：

```
.intro {
  font-weight: bold;
  font-size: 1.2em;
}
```

有了这个声明，我们只需在 styleHeaderSiblings()函数里把紧跟在一级标题之后的那个元素的 class 属性设置为 intro 就可以达到同样的目的。

可以用 setAttribute()方法来做这件事：

```
elem.setAttribute("class","intro");
```

更简单的办法是对 className 属性进行刷新。className 属性是一个可读/可写的属性，而且凡是元素节点都有这个属性。

我们可以用 className 属性检索某给定元素的 class 属性值：

```
element.className
```

用 className 属性和赋值操作符设置某给定元素的 class 属性：

```
element.className = value
```

下面是利用 className 属性编写出来的 styleHeaderSiblings()函数，它在设置样式信息时不需要直接与 style 属性打交道：

```
function styleHeaderSiblings() {
  if (!document.getElementsByTagName) return false;
  var headers = document.getElementsByTagName("h1");
  for (var i=0; i<headers.length; i++) {
    var elem = getNextElement(headers[i].nextSibling);
    elem.className = "intro";
  }
}
```

现在，当我们想改变紧跟在一级标题之后的那个元素的样式时，我们只需在 CSS 里修改.intro 类的样式声明即可：

```
.intro {
  font-weight: bold;
  font-size: 1.4em;
}
```

这个技巧只有一个不足：通过 className 属性设置某个元素的 class 属性值将替换（而不是追加）该元素现有的一切 class 设置：

```
<h1>Man bites dog</h1>
<p class="disclaimer">This is not a true story</p>
```

如果对包含着以上 HTML 内容的文档使用 styleHeaderSiblings()函数，那个“文本段”元素的 class 属性将从 disclaimer 被替换为 intro，而这里实际需要的是“追加”效果——class 属性应该变成 disclaimer intro，也就是 disclaimer 和 intro 两种样式的叠加。

我们可以利用字符串拼接操作把新的 class 设置值追加到 className 属性上去（请注意，intro 的第一个字符是空格），如下所示：

```
elem.className += " intro";
```

不过，这个字符串拼接操作只在确实存在着现有 class 设置值的情况下才有必要执行。如果不存在任何现有的 class 设置值，直接对 className 属性进行赋值就可以了。

在需要给某个元素追加新的 class 设置值时，我们可以按照以下步骤操作：

(1) className 属性的值是否为 null。

(2) 如果是，把新的 class 设置值直接赋值给 className 属性。

(3) 如果不是，把一个空格和新的 class 设置值追加到 className 属性上去。

我们可以把以上步骤实现为一个函数 addClass。这个函数带两个参数：其一是将获得新 class 设置值的元素（element）；其二是新的 class 设置值（value）：

```
function addClass(element,value) {
  if (!element.className) {
    element.className = value;
  } else {
    newClassName = element.className;
    newClassName+= " ";
    newClassName+= value;
    element.className = newClassName;
  }
}
```

在 styleHeaderSiblings()函数里调用 addClass()函数：

```
function styleHeaderSiblings() {
  if (!document.getElementsByTagName) return false;
  var headers = document.getElementsByTagName("h1");
  for (var i=0; i<headers.length; i++) {
    var elem = getNextElement(headers[i].nextSibling);
    addClass(elem,"intro");
  }
}
```

我们还可以对刚才编写的 stripeTables()函数做类似的改进。这个函数现在是通过直接改变偶数表格行的背景颜色来实现斑马线效果的：

```
function stripeTables() {
  if (!document.getElementsByTagName) return false;
  var tables = document.getElementsByTagName("table");
  for (var i=0; i<tables.length; i++) {
    var odd = false;
    var rows = tables[i].getElementsByTagName("tr");
    for (var j=0; j<rows.length; j++) {
      if (odd == true) {
        rows[j].style.backgroundColor = "#ffc";
        odd = false;
      } else {
        odd = true;
      }
    }
  }
}
```

先在 format.css 文件里增加一条对应于 class="odd"的样式声明：

```
.odd {
  background-color: #ffc;
}
```

然后修改 stripeTables()函数，让它通过调用 addClass()函数的办法来实现同样的效果：

```
function stripeTables() {
  if (!document.getElementsByTagName) return false;
  var tables = document.getElementsByTagName("table");
  for (var i=0; i<tables.length; i++) {
    var odd = false;
    var rows = tables[i].getElementsByTagName("tr");
    for (var j=0; j<rows.length; j++) {
      if (odd == true) {
        addClass(rows[j],"odd");
        odd = false;
      } else {
        odd = true;
      }
    }
  }
}
```

基于 className 属性的新解决方案与基于 style 属性的老解决方案有相同的最终结果。但新解决方案是通过 CSS 而不是 DOM 去设置样式信息的。新解决方案里的 JavaScript 函数刷新的是 className 属性，style 属性不受任何影响。新解决方案使得网页的表示层和行为层分离得更加彻底了。

对函数进行抽象化

函数都工作的很好，我们完全可以让它们就保持现在这样。不过，如果能再对它们做一些小

小的改进的话，它们的适用范围将大大增加。把某种非常具体的东西改进为一种较为通用的东西的过程叫作抽象化（abstraction）。

如果仔细看看 styleHeaderSiblings()函数，我们就会发现它仅适用于 h1 元素，而且 classNeme 属性值 intro 也是硬编码在函数代码里的：

```
function styleHeaderSiblings() {
  if (!document.getElementsByTagName) return false;
  var headers = document.getElementsByTagName("h1");
  for (var i=0; i<headers.length; i++) {
    var elem = getNextElement(headers[i].nextSibling);
    addClass(elem,"intro");
  }
}
```

如果把这些具体的值转换为这个函数的参数，我们就可以让它成为一个更通用的函数。我决定把改进后的新函数命名为 styleElementSibling 并给它增加两个参数——tag 和 theclass：

```
function styleElementSiblings(tag,theclass)
```

接下来，把函数代码中的字符串“h1”全部替换为参数变量 tag，再把字符串“intro”全部替换为参数变量 theclass。为了增加代码的可读性，我还顺便把原来的 headers 变量替换成了 elems 变量：

```
function styleElementSiblings(tag,theclass) {
  if (!document.getElementsByTagName) return false;
  var elems = document.getElementsByTagName(tag);
  for (var i=0; i<elems.length; i++) {
    var elem = getNextElement(elems[i].nextSibling);
    addClass(elem,theclass);
  }
}
```

现在，如果把字符串值“h1”和“intro”作为参数传递给这个新函数，就可以获得原来的效果：

```
styleElementSiblings("h1","intro");
```

一般来说，每当我们发现自己可以像上面这样对某个函数进行抽象化时，就应该那样去做：这可以为今后的工作提供许多方便。比如说，我们今后或许会遇到需要对另一种元素和/或另一个 className 属性值进行类似于对 styleHeaderSiblings()函数这样的处理的情况。如果真是那样，一个像 styleElementSiblings()这样更通用的函数将帮上我们的大忙。

9.5　小结

在这一章里，我们探讨了 DOM 的一种全新应用模式。我们此前介绍的 DOM 方法和属性分别隶属于 DOM Core 和 HTML-DOM 等细分领域。本章介绍的 CSS-DOM 技术针对的是如何提取（读）和设置（写）style 对象的各种属性，而 style 对象本身又是 XHTML 文档中的每个元

素节点都具备的属性。

style 属性很有用，但它的最大不足是我们无法通过这一属性提取到通过外部 CSS 设置的样式信息。但我们仍可以利用 style 属性去改变各种 HTML 元素的样式，进而改变它们在浏览器里的呈现效果。这在我们无法或是难以通过 CSS 去完成样式设置任务的场合将非常有用。在挑选解决方案时，只要有可能，我们应该选择“刷新 className 属性值”而不是“直接刷新 style 对象的有关属性”。

在这一章里，我们向大家介绍了以下几种 CSS-DOM 技术的具体应用示例：

- 根据有关元素在节点树里的位置去设置它们的样式信息（styleHeaderSiblings()函数）
- 通过遍历一个节点集合去设置有关元素的样式信息（stripeTables()函数）
- 在对事件做出响应时设置有关元素的样式信息（highlightRows()函数）

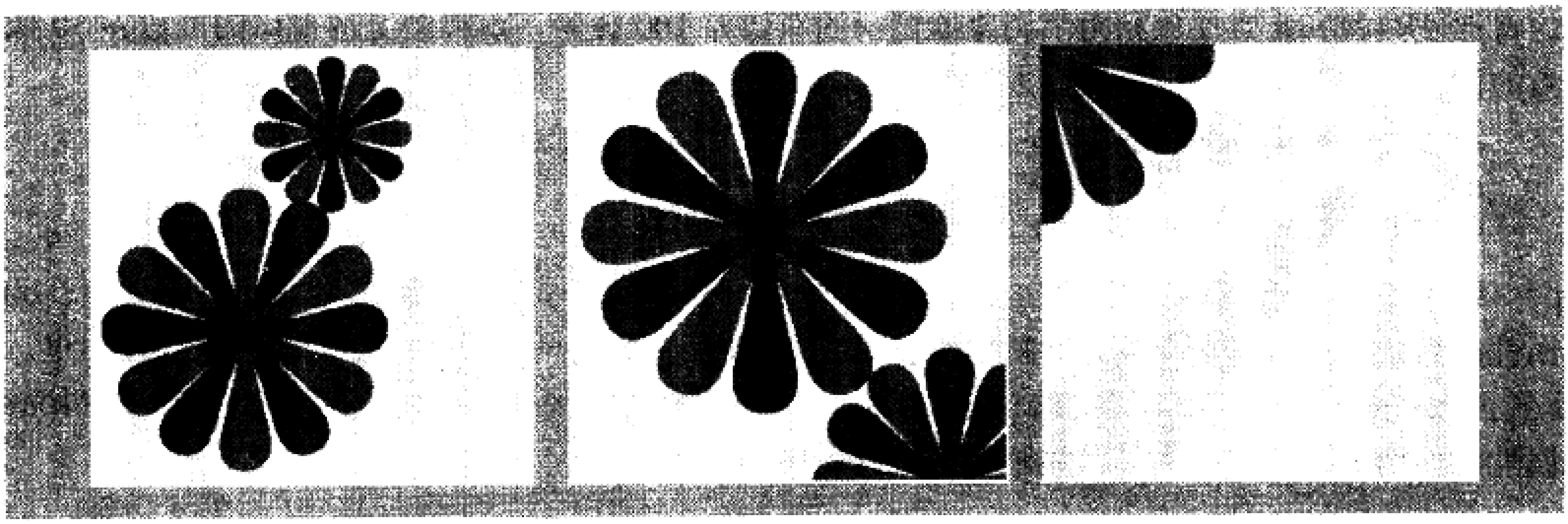
这几种应用都属于用 JavaScript 入侵 CSS 领地的情况，而我们这么做的理由不外乎两点：其一是 CSS 无法让我们找到我们想要处理的目标元素；其二是用 CSS 寻找目标元素的办法还未得到广泛的支持。或许，未来的 CSS 技术能够让我们远离这种“不务正业”的 DOM 脚本编程技术。

不过，有一种应用大概是 CSS 永远也无法与 DOM 竞争的：JavaScript 脚本可以周期性地重复执行一组操作；通过周期性地改变样式信息，我们就可以实现出在只使用 CSS 的情况下根本不可能实现的效果。

在下一章里，你们将会看到一个那样的例子。我们将编写一个能够随着时间的推移而不断刷新(X)HTML 元素位置的函数。简单地说，我们将用 JavaScript 代码实现出动画效果。

第 10 章

用 JavaScript 实现动画效果



本章内容

- ❑ 何为动画
- ❑ 用动画丰富网页的浏览效果
- ❑ 让动画效果更流畅

在这一章里，你将看到 CSS-DOM 技术最富于动感的应用之一：让网页上的元素动起来。

10.1 何为动画

在上一章里，我向大家介绍了如何利用 DOM 技术去设置和修改某个文档的样式信息。用 JavaScript 添加样式信息可以节约许多时间和精力，但总的来说，CSS 仍是完成这类任务的最佳工具。

不过，有一个应用领域是 CSS 无能为力的：如果我们想随着时间的变化而不断改变某个元素的样式，则只能使用 JavaScript。JavaScript 脚本能够按照预定的时间间隔重复调用一个函数，而这意味着我们可以随着时间的推移而不断改变某个元素的样式。

动画是样式随时间变化的完美例子之一。简单地说，动画就是让元素的位置随着时间而不断地发生变化。

10.1.1 位置

网页元素在浏览器窗口里的位置是一种表示性的信息。因此，位置信息通常是由 CSS 负责设置的。下面这些 CSS 示例代码对某个元素在网页上的位置做出了规定：

```
element {
  position: absolute;
  top: 50px;
  left: 100px;
}
```

这将把 *element* 元素摆放到距离浏览器窗口的左边界 100 像素，距离浏览器窗口的上边界 50 像素的位置上。下面是实现同样效果的 DOM 代码：

```
element.style.position = "absolute";
element.style.left = "100px";
element.style.top = "50px";
```

position 属性的合法取值有 static、fixed、relative 和 absolute 四种。static 是 position 属性的默认值，意思是有关元素将按照它们在 HTML 文档里的先后顺序出现在浏览器窗口里。relative 的含义与 static 相似，区别是 position 属性等于 relative 的元素还可以（在 float 属性的作用下）从文档的正常显示顺序里脱离出来。

如果把某个元素的 position 属性设置为 absolute，我们就可以把它摆放到它的“容器”（所谓的“容器”通常就是文档本身）里的任何位置。这个元素在原始 HTML 文档里出现的位置对此没有任何影响，它的显示位置将由 top、left、right 和 bottom 等属性决定。在设置这些属性时，我们可以使用“像素”（比如 100px）或毫米、厘米等作为度量单位。

把某个元素的 top 属性设置为（比如说）100px 将把该元素摆放到距文档顶边界 100“像素”的位置，而把某个元素的 bottom 属性设置为 100px 将把该元素摆放到距文档底边界 100“像素”的位置。类似地，把某个元素的 left 或 right 属性设置为 100px 将把该元素摆放到距文档左边界或右边界 100“像素”的位置。为防止它们发生冲突，最好是只使用 top 或 bottom 属性之一；left 或 right 属性也是如此。

把文档里的某个元素摆放到一个特定的位置是很容易的事。我们不妨假设有一个下面这样的元素：

```
<p id="message">Whee!</p>
```

于是，可以用一个如下所示的 JavaScript 函数来设置这个元素的位置：

```
function positionMessage() {
  if (!document.getElementById) return false;
  if (!document.getElementById("message")) return false;
  var elem = document.getElementById("message");
  elem.style.position = "absolute";
  elem.style.left = "50px";
```

```
    elem.style.top = "100px";
  }
```

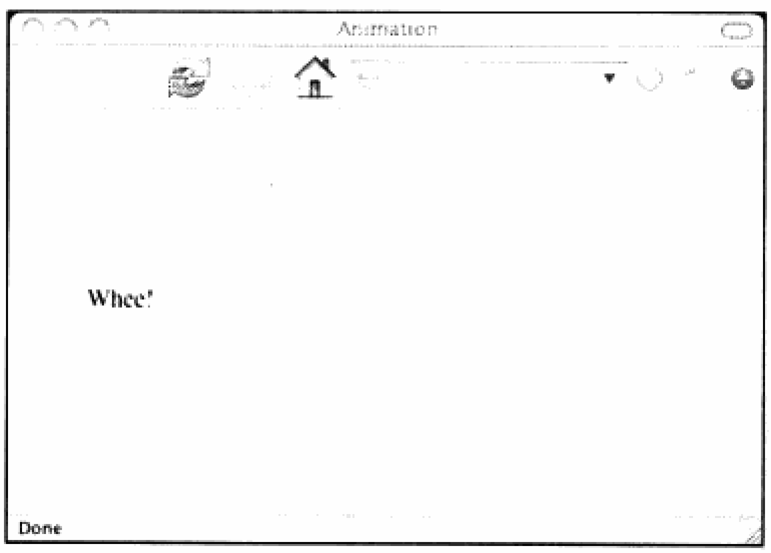
在页面加载时调用这个 positionMessage()函数，将把这段文本摆放到距浏览器窗口的左边界 50 像素、距浏览器窗口的顶边界 100 像素的位置：

```
window.onload = positionMessage;
```

不过，把函数绑定到 window.onload 事件处理函数的工作最好是用 addLoadEvent()函数来完成，如下所示：

```
function addLoadEvent(func) {
  var oldonload = window.onload;
  if (typeof window.onload != 'function') {
    window.onload = func;
  } else {
    window.onload = function() {
      oldonload();
      func();
    }
  }
}
addLoadEvent(positionMessage);
```

下面是按 position="absolute"的情况来摆放这个元素的效果。



改变某个元素的位置也很简单，只要执行一个函数去改变这个元素的 style.top 或 style.left 等属性就行了：

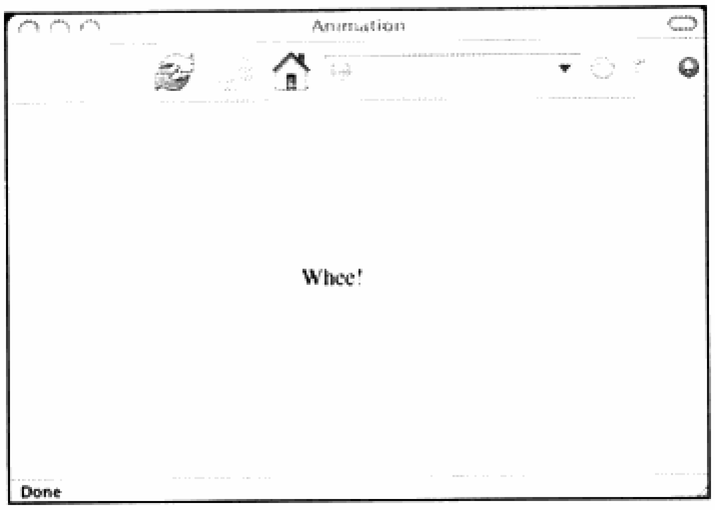
```
function moveMessage() {
  if (!document.getElementById) return false;
  if (!document.getElementById("message")) return false;
  var elem = document.getElementById("message");
  elem.style.left = "200px";
}
```

编写一个这样的函数并不难，问题是该如何去调用这个函数呢？如果让 moveMessage()函数在页

面加载时运行，这个元素的位置将立刻发生变化——由 positionMessage()函数给出的原始位置会被立刻覆盖：

```
addLoadEvent(positionMessage);
addLoadEvent(moveMessage);
```

请看，这个元素的显示位置立刻发生了变化。



元素的显示位置立刻发生变化并不是我们想要的动画效果。要想获得真正的动画效果，必须让元素的位置随着时间不断地发生变化。

10.1.2 时间

导致元素的显示位置立刻发生变化的根源是 JavaScript 太有效率了：函数一个接一个地执行，其间根本没有我们所能察觉的间隔。为了实现动画效果，我们必须“创造”出时间间隔来；而这正是我们将要探讨的问题。

1. setTimeout()函数

JavaScript 函数 setTimeout()能够让某个函数在经过一段预定的时间之后才开始执行。这个函数带有两个参数：第一个参数是一个字符串，其内容是将要执行的那个函数的名字；第二个参数是一个数值，它以毫秒为单位设定了需要经过多长时间才开始执行由第一个参数所给出的函数：

```
setTimeout("function",interval)
```

在绝大多数时候，把这个函数调用赋值给一个变量将是个好主意：

```
variable = setTimeout("function",interval)
```

如果你想能够取消某个正在排队等待执行的函数，就必须像上面这样把相应的函数调用赋值给一个变量。我们可以用一个名为 clearTimeout 的函数来取消“等待执行”队列里的某个尚未开始执行的函数。这个函数需要我们向它传递一个参数——与某个 setTimeout()函数调用相关联的变量：

```
clearTimeout(variable)
```

我将修改 positionMessage()函数，让它在 5 秒（或者说 5000 毫秒）之后才去调用 moveMessage()函数：

```
function positionMessage() {
  if (!document.getElementById) return false;
  if (!document.getElementById("message")) return false;
  var elem = document.getElementById("message");
  elem.style.position = "absolute";
  elem.style.left = "50px";
  elem.style.top = "100px";
  movement = setTimeout("moveMessage()",5000);
}
```

positionMessage()函数仍将在页面加载时得到执行：

```
addLoadEvent(positionMessage);
```

这样一来，那条消息将先出现在它的原始位置上，然后在 5 秒之后才向右“跳跃”150 像素。

在那 5 秒钟的等待时间里，我可以随时使用下面这条语句取消这一“跳跃”行为：

```
clearTimeout(movement);
```

movement 变量对应着在 positionMessage()函数里定义的 setTimeout()调用。它是一个全局变量——我在声明它时未使用 JavaScript 关键字 var。这意味着那个“跳跃”行为可以在 positionMessage()函数以外的地方被取消。

2. 时间递增量

把某个元素在 5 秒钟之后向右移动 150 像素的显示效果称为动画实在有点儿勉强。真正的动画效果是一个渐变的过程，元素应该从出发点逐步地移动到目的地而不是从出发点一下子跳跃到目的地。

我将修改 moveMessage()函数，让元素的移动以渐变的方式发生。下面是新 moveMessage()函数的逻辑：

(1) 获得元素的当前位置。
(2) 如果元素已经到达它的目的地，则退出这个函数。
(3) 如果元素尚未到达它的目的地，则把它向目的地移近一点儿。
(4) 经过一段时间间隔之后从步骤 1 开始重复上述步骤。

第一步是确定元素的当前位置。这一点可以通过查询元素的 style.top 和 style.left 等位置属性做到。我将把 style.top 和 style.left 属性的值分别赋值给变量 xpos 和 ypos：

```
var xpos = elem.style.left;
var ypos = elem.style.top;
```

当 moveMessage()函数在 positionMessage()函数之后被调用时，xpos 变量将被赋值为 50px，ypos 变量将被赋值为 100px。我遇到了一点儿小麻烦：这两个值都是字符串，而接下来的代码需要用到许多算术比较操作符。我需要的是数值，不是字符串。

- parseInt()函数

JavaScript 函数 parseInt()可以把字符串里的数值信息提取出来。如果把一个以数字开头的字符串传递给这个函数，它将返回一个数值：

```
parseInt(string)
```

下面是一个例子：

```
parseInt("39 steps");
```

这个函数调用将返回数值“39”（不包括引号）。

parseInt()函数的返回值永远是整数。如果需要提取的是带小数点的数值（也就是浮点数），就应该使用相应的 parseFloat()函数：

```
parseFloat(string)
```

我在 moveMessage()函数里将只与整数打交道，所以使用 parseInt()函数：

```
var xpos = parseInt(elem.style.left);
var ypos = parseInt(elem.style.top);
```

parseInt()函数将把字符串“50px”转换为整数 50，把字符串“100px”转换为整数 100。现在，xpos 和 ypos 变量分别包含整数 50 和 100。

在 moveMessage()函数里，接下来的几个步骤需要用到大量的算术比较操作符。

首先，进行一次“等于”比较：我需要知道变量 xpos 和 ypos 的值是否等于目的地那里的“左”位置和“上”位置。如果是，退出这个函数。“等于”比较的操作符是由两个等号构成的“==”。

注意　单个等号（=）是赋值操作符，不是比较操作符。

```
if (xpos == 200 && ypos == 100) {
  return true;
}
```

上面这条 if 语句的含义是：如果 id="mesage"的那个元素还没有到达它的目的地，则继续执行。

接下来，根据 mesage 元素的当前位置及其目的地位置的关系去刷新变量 xpos 和 ypos 的值。我想通过刷新这两个变量把 mesage 元素移动到一个距离其目的地更近的地方。

如果变量 xpos 的值小于目的地那里的“左”位置，给它加 1：

```
if (xpos < 200) {
  xpos++;
}
```

如果变量 xpos 的值大于目的地那里的“左”位置，给它减 1：

```
if (xpos > 200) {
  xpos--;
}
```

对变量 ypos 的值进行同样的刷新，但这次的比较参照物是目的地那里的“上”位置：

```
if (ypos < 100) {
  ypos++;
}
if (ypos > 100) {
  ypos--;
}
```

现在，知道为何需要把变量 xpos 和 ypos 从字符串转换为数值了吧：因为我需要用“大于”和“小于”操作符把它们和一些数值进行比较，并根据比较的结果对它们进行相应的刷新。

接下来，需要把变量 xpos 和 ypos 的值传递给 mesage 元素的 style 属性。具体地说，我将把字符串“px”追加到这两个变量值的末尾并把它们赋值给 left 和 top 属性：

```
elem.style.left = xpos + "px";
elem.style.top = ypos + "px";
```

最后，我需要在经过一个短暂的停顿之后重复执行这个函数（从一个函数的内部调用这个函数本身叫作递归调用）。我决定把停顿时间设置为百分之一秒，也就是 10 毫秒：

```
movement = setTimeout("moveMessage()",10);
```

下面是 moveMessage()函数的代码清单：

```
function moveMessage() {
  if (!document.getElementById) return false;
  if (!document.getElementById("message")) return false;
  var elem = document.getElementById("message");
  var xpos = parseInt(elem.style.left);
  var ypos = parseInt(elem.style.top);
  if (xpos == 200 && ypos == 100) {
    return true;
  }
  if (xpos < 200) {
    xpos++;
  }
  if (xpos > 200) {
    xpos--;
  }
  if (ypos < 100) {
```

```
    ypos++;
  }
  if (ypos > 100) {
    ypos--;
  }
  elem.style.left = xpos + "px";
  elem.style.top = ypos + "px";
  movement = setTimeout("moveMessage()",10);
}
```

这个函数将使得 message 元素以每次一“像素”的方式在浏览器窗口里移动。一旦这个元素的 top 和 left 属性同时等于 100px 和 200px，这个函数就停止执行。这可是实实在在的动画效果——虽然它没有什么实际的意义。稍后，我们将利用同样的原理实现一个更有实用价值的例子。

10.1.3 抽象化

刚才编写的 moveMessage()函数只能完成一项非常特定的任务。它将把一个特定的元素移动到一个特定的位置，两次移动之间的停顿时间也是一段固定的长度；所有这些信息都是硬编码在函数代码里的：

```
function moveMessage() {
  if (!document.getElementById) return false;
  if (!document.getElementById("message"))
return false;
  var elem = document.getElementById("message");
  var xpos = parseInt(elem.style.left);
  var ypos = parseInt(elem.style.top);
  if (xpos == 200 && ypos == 100) {
    return true;
  }
  if (xpos < 200) {
    xpos++;
  }
  if (xpos > 200) {
    xpos--;
  }
  if (ypos < 100) {
    ypos++;
  }
  if (ypos > 100) {
    ypos--;
  }
  elem.style.left = xpos + "px";
  elem.style.top = ypos + "px";
  movement = setTimeout("moveMessage()",10);
}
```

如果把这些常数都改为变量，这个函数的灵活性和适用范围就会大大提高。通过对 moveMessage()

函数进行抽象化，我们将得到一个用途更广和用起来更方便的函数。

我决定把新函数命名为 moveElement。与 moveMessage()函数不同，新函数的参数将会有多个。下面是每次调用那个新函数时可以加以变化的内容：

(1) 我们打算移动的那个元素的 ID。
(2) 该元素的目的地的“左”位置。
(3) 该元素的目的地的“上”位置。
(4) 两次移动之间的停顿时间。

这些参数都应该有一个有含义的名字：

(1) elementID
(2) final_x
(3) final_y
(4) interval

定义 moveElement()函数的第一步是声明它的各个参数：

```
function moveElement(elementID,final_x,final_y,interval) {
```

我将用这些变量把硬编码在 moveMessage()函数里的有关常数全部替换掉。首当其冲的是 moveMessage()函数开头部分的几条语句，如下所示：

```
if (!document.getElementById) return false;
if (!document.getElementById("message")) return false;
var elem = document.getElementById("message");
```

把上面这几条语句里的 getElementById("message")全部替换为 getElementById(elementID)：

```
if (!document.getElementById) return false;
if (!document.getElementById(elementID)) return false;
var elem = document.getElementById(elementID);
```

接下来的两行语句用不着修改。它们负责把给定元素的 left 和 top 属性转换为数值，并把转换结果分别赋值给变量 xpos 和 ypos：

```
var xpos = parseInt(elem.style.left);
var ypos = parseInt(elem.style.top);
```

接下来，检查给定元素是否已经到达目的地。在 moveMessage()函数里，目的地的坐标值是 200（“左”位置）和 100（“上”位置）

```
if (xpos == 200 && ypos == 100) {
  return true;
}
```

在 moveElement()函数里，目的地的坐标值将由变量 final_x 和 final_y 提供：

```
if (xpos == final_x && ypos == final_y) {
  return true;
}
```

再往后是对变量 xpos 和 ypos 进行刷新的几条语句。如果变量 xpos 的值小于目的地的“左”位置，给它加 1。

原来的目的地的“左”位置是硬编码在函数代码里的常数 200：

```
if (xpos < 200) {
  xpos++;
}
```

现在的目的地的“左”位置由变量 final_x 提供：

```
if (xpos < final_x) {
  xpos++;
}
```

类似地，如果变量 xpos 的值大于目的地的“左”位置，xpos 变量的值减 1：

```
if (xpos > final_x) {
  xpos--;
}
```

对负责刷新变量 ypos 的语句做同样的修改。如果变量 ypos 的值小于 final_y，给它加 1；如果它大于 final_y，给它减 1：

```
if (ypos < final_y) {
  ypos++;
}
if (ypos > final_y) {
  ypos--;
}
```

接下来的两行语句用不着修改。它们负责把字符串"px"追加到变量 xpos 和 ypos 的末尾，并将其赋值给 elem 元素的 left 和 top 样式属性：

```
elem.style.left = xpos + "px";
elem.style.top = ypos + "px";
```

最后，在经过一段适当的时间间隔之后再次调用同一个函数。在 moveMessage()函数里，这个环节相当简单：每隔 10 毫秒调用一次 moveMessage()函数：

```
movement = setTimeout("moveMessage()",10);
```

在 moveElement()函数里，事情变得有一点儿复杂。这次，在发出一个递归调用时，我们还需要传递一些参数：elementID、final_x、final_y 和 interval。所有这些将形成一个如下所示的字符串：

```
"moveElement('"+elementID+"',"+final_x+","+final_y+","+interval+")"
```

字符串拼接操作实在不少！与其把一个这么长的字符串直接插入到 setTimeout()函数里去，不如先把这个字符串赋值给 repeat 变量：

```
var repeat =
➥ "moveElement('"+elementID+"',"+final_x+","+final_y+","+interval+")";
```

现在，我们只需把 repeat 变量插入到 setTimeout()函数里作为它的第一个参数就行了。第二个参数是再次调用第一个参数所指定的函数之前需要等待的时间间隔。这个间隔在 moveMessage()函数里被硬编码为 10 毫秒，它现在将由变量 interval 提供：

```
movement = setTimeout(repeat,interval);
```

用一个右花括号结束这个函数：

```
}
```

下面是 moveElement()函数的代码清单：

```
function moveElement(elementID,final_x,final_y,interval) {
  if (!document.getElementById) return false;
  if (!document.getElementById(elementID)) return false;
  var elem = document.getElementById(elementID);
  var xpos = parseInt(elem.style.left);
  var ypos = parseInt(elem.style.top);
  if (xpos == final_x && ypos == final_y) {
    return true;
  }
  if (xpos < final_x) {
    xpos++;
  }
  if (xpos > final_x) {
    xpos--;
  }
  if (ypos < final_y) {
    ypos++;
  }
  if (ypos > final_y) {
    ypos--;
  }
  elem.style.left = xpos + "px";
  elem.style.top = ypos + "px";
  var repeat =
➥ "moveElement('"+elementID+"',"+final_x+","+final_y+","+interval+")";
  movement = setTimeout(repeat,interval);
}
```

把 moveElement()函数保存为 moveElement.js 文件。把这个文件放入 scripts 文件夹，别忘了把 addLoadEvent.js 文件也放到那里。

现在，我们来测试一下这个函数。

首先，创建一个名为 message.html 的文档，让它包含一个 id 属性值是 message 的文本段：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Message</title>
</head>
<body>
  <p id="message">Whee!</p>
</body>
</html>
```

要想看到动画效果，我们必须先把那个文本段摆放到出发点位置上。编写一个名为 positionMessage.js 的 JavaScript 文件。在 positionMessage()函数的末尾，调用 moveElement()函数：

```
function positionMessage() {
  if (!document.getElementById) return false;
  if (!document.getElementById("message")) return false;
  var elem = document.getElementById("message");
  elem.style.position = "absolute";
  elem.style.left = "50px";
  elem.style.top = "100px";
  moveElement("message",200,100,10);
}
addLoadEvent(positionMessage);
```

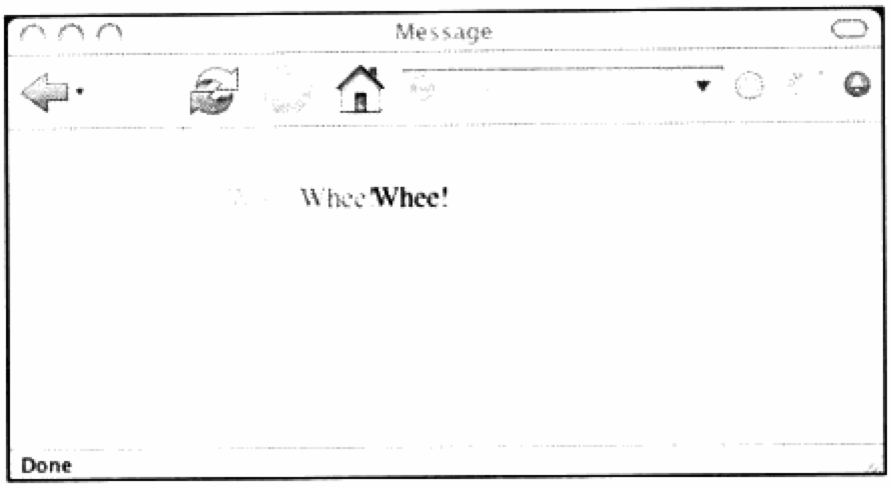
上面这段代码中的 moveElement()函数调用语句将把字符串值“message”传递给 elementID 参数，把数值 200 传递给 final_x 参数，把数值 100 传递给 final_y 参数，把数值 10 传递给 interval 参数。

scripts 文件夹现在包含三个文件：addLoadEvent.js、positionMessage.js 和 moveElement.js。我们需要在 message.html 文档里插入一些<script>标签来引用这几个脚本文件，如下所示：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Message</title>
  <script type="text/javascript" src="scripts/addLoadEvent.js">
  </script>
  <script type="text/javascript" src="scripts/positionMessage.js">
  </script>
  <script type="text/javascript" src="scripts/moveElement.js">
  </script>
```

```
</head>
<body>
  <p id="message">Whee!</p>
</body>
</html>
```

现在，把 message.html 文档加载到一个 Web 浏览器里就可以看到我们所实现的动画效果了：那个元素将在浏览器窗口里横向移动。



至此，一切都进行得很顺利。moveElement()函数与 moveMessage()函数的效果完全一样。不过，因为我们已经对这个函数进行过抽象化处理，所以现在可以把任意的参数传递给它。比如说，如果改变参数 final_x 和 final_y 的值，就可以改变动画的移动方向；如果改变参数 interval 的值，就可以改变动画的移动速度：

```
function moveElement(elementID,final_x,final_y,interval)
```

在 positionMessage.js 文件里修改 positionMessage()函数的最后一行，让这三个值发生点儿变化：

```
function positionMessage() {
  if (!document.getElementById) return false;
  if (!document.getElementById("message")) return false;
  var elem = document.getElementById("message");
  elem.style.position = "absolute";
  elem.style.left = "50px";
  elem.style.top = "100px";
  moveElement("message",125,25,20);
}
addLoadEvent(positionMessage);
```

在 Web 浏览器里刷新 message.html 文件，就可以看到新的动画效果了：那个元素现在将斜向移动，移动的速度也变慢了：

还可以改变 moveElement()函数的 elementID 参数值：

```
function moveElement(elementID,final_x,final_y,interval)
```



在 message.html 文件里增加一个新元素，把它的 id 属性设置为 message2：

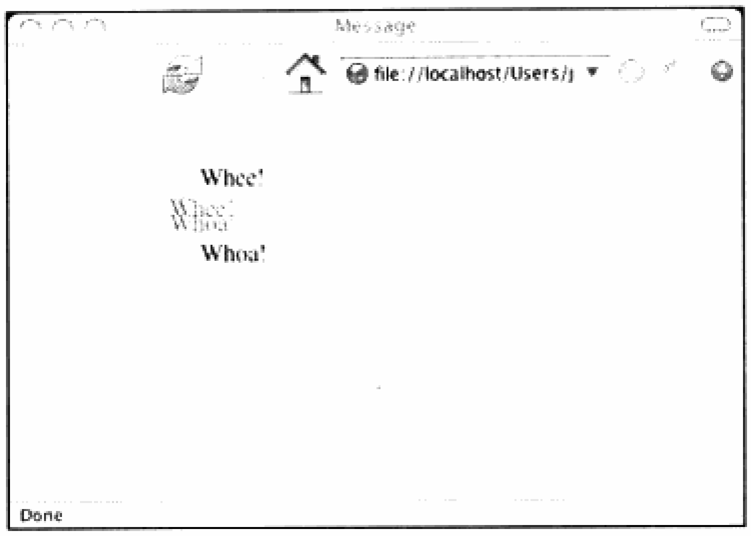
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Message</title>
  <script type="text/javascript" src="scripts/addLoadEvent.js">
  </script>
  <script type="text/javascript" src="scripts/positionMessage.js">
  </script>
  <script type="text/javascript" src="scripts/moveElement.js"></script>
</head>
<body>
  <p id="message">Whee!</p>
  <p id="message2">Whoa!</p>
</body>
</html>
```

现在，在 positionMessage.js 文件里增加一些代码。先为 message2 元素设定一个初始位置，然后增加一条 moveElement()函数调用语句——把 message2 作为它的第一个参数传递：

```
function positionMessage() {
  if (!document.getElementById) return false;
  if (!document.getElementById("message")) return false;
  var elem = document.getElementById("message");
  elem.style.position = "absolute";
  elem.style.left = "50px";
  elem.style.top = "100px";
  moveElement("message",125,25,20);
  if (!document.getElementById("message2")) return false;
  var elem = document.getElementById("message2");
  elem.style.position = "absolute";
  elem.style.left = "50px";
  elem.style.top = "50px";
  moveElement("message2",125,75,20);
}
addLoadEvent(positionMessage);
```

在 web 浏览器里刷新 message.html 文件就可以看到新的动画效果了：两个元素将沿着不同的方向同时移动。

在这两个例子里，所有工作都是由 moveElement()函数完成的。只需简单地改变一下传递给这个函数的参数值，我们就可以用它实现出不同的动画效果。这正是用参数变量代替硬编码常数的最大好处之一。



10.2 实用的动画

有了 moveElement()函数，我们就有了一个通用的、可以沿任意方向移动多个页面元素的函数。从程序设计的角度看，这会给人留下相当深刻的印象；但从实用的角度看，它的意义似乎并不大。

网页上的动画元素不仅容易引起访问者的反感，还容易导致各种各样的可访问性问题。W3C 在它们的 Web Content *Accessibility* Guidelines（Web 内容可访问性指南）7.2 节里给出了这样的建议：“如果你不清楚最终用户所使用的软件是否允许用户“冻结”移动着的元素，就应该避免让元素在页面里四处移动。[优先级 2]如果你的页面上有可移动的内容，就应该在脚本或插件里提供一种让用户能够冻结这种移动或动态刷新行为的机制。”

这里的关键问题是用户能不能对可移动内容进行控制。在解决了这个问题的基础上，根据用户的操作行为去移动某个页面元素往往会是一种很不错的网页充实手段。我很愿意向大家介绍一个能够实现这种内容充实效果的例子。

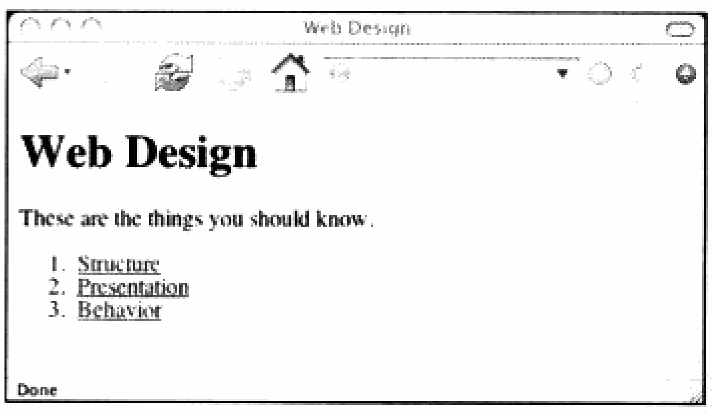
10.2.1 问题的提出

我有一个包含一系列链接的网页。当用户把鼠标指针悬停在其中的某个链接上时，我想用一种与众不同的方式来告诉用户这个链接将把他们带到何方。简单地说，我想让用户看到一张预览图片。

这个网页的基本文档是 list.html 文件，下面是它的代码清单：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Web Design</title>
</head>
<body>
  <h1>Web Design</h1>
  <p>These are the things you should know.</p>
  <ol id="linklist">
    <li>
      <a href="structure.html">Structure</a>
    </li>
    <li>
      <a href="presentation.html">Presentation</a>
    </li>
    <li>
      <a href="behavior.html">Behavior</a>
    </li>
  </ol>
</body>
</html>
```

这个网页里的每个链接分别指向一个介绍相关网页设计技巧的页面。这些链接的屏显标识文本已经足以让用户对链接目标页面的内容有一个大致的了解。



这个网页本身已经足够完美——我的意思是说，该有的东西都已经有了。在此基础上，我认为给目标文档增加一种视觉提示会让这个网页更有吸引力。

从某种意义上讲，这个案例与我们在本书前面的有关章节里实现的“JavaScript 美术馆”颇为相似：它们都包含着一系列链接，我想对它们做的改进都是显示一张图片。但我这一次是想在 onmouseover 事件处理函数（请注意，不是 onclick 事件处理函数）被触发时显示一张图片。

我将沿用“JavaScript 美术馆”案例中的脚本——只需把每个链接上的事件处理函数从 onclick 改为 onmouseover。这个解决方案完全行得通，但我觉得图片显示得还不够流畅：当用户第一次把鼠标指针悬停在某个链接上时，新图片将被加载过去。即使是在一个高速的网络连接

上，这多少也需要花费点儿时间，而我希望这个网页能够对此立刻做出响应。

10.2.2 问题的解决

如果我为每个链接分别准备一张预览图片的话，在切换显示那些图片的时候还是会有一些延迟。事实上，简单地切换显示那些图片并不是我想要的效果。我想要的是一种更快更好的东西。

下面是我将做的事情：

- ❑ 把所有的预览图片生成为一张“集体照”形式的组合图片。
- ❑ 隐藏这张“集体照”图片的绝大部分。
- ❑ 当用户把鼠标指针悬停在某个链接的上方时，只显示这张“集体照”图片的一个部分。

我已经制作出了一张这样的“集体照”图片，它由三张预览图片和一张默认图片构成，如下所示。



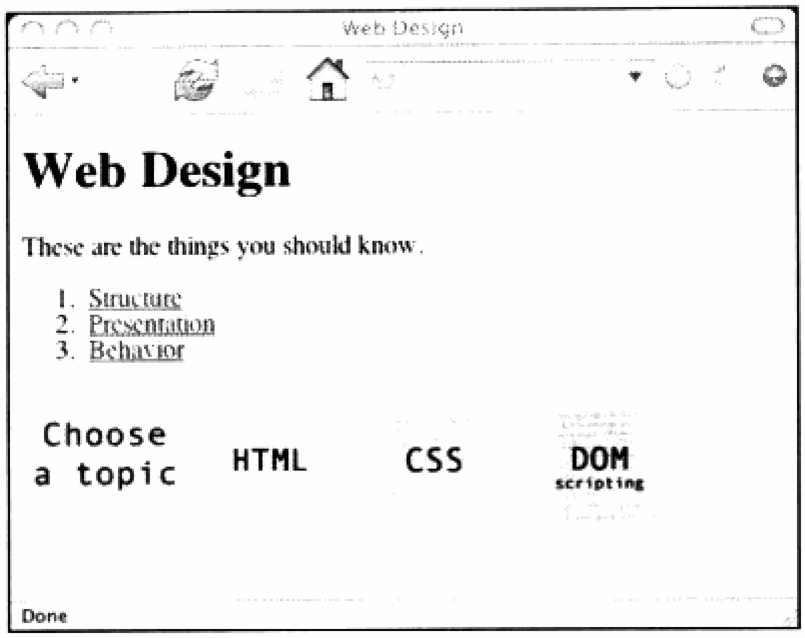
这个图片的文件名是 topics.gif。它的宽度是 400 像素，高度是 100 像素。

我将把 topics.gif 图片插入到 list.html 文档里，并把这个图片元素的 id 属性设置为 preview：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Web Design</title>
</head>
<body>
  <h1>Web Design</h1>
  <p>These are the things you should know.</p>
  <ol id="linklist">
    <li>
      <a href="structure.html">Structure</a>
    </li>
    <li>
      <a href="presentation.html">Presentation</a>
    </li>
    <li>
      <a href="behavior.html">Behavior</a>
    </li>
  </ol>
  <img src="topics.gif" alt="building blocks of web design"
➥ id="preview" />
```

```
</body>
</html>
```

下面是带着那些链接和那张“集体照”图片的网页显示效果。



现在，“集体照”图片的全部画面都是可见的。我想每次只让这个图片的某个 100×100 像素的部分出现在浏览器窗口里。我无法用 JavaScript 做到这一点，但可以用 CSS 来做这件事。

10.2.3　CSS

如果某个元素的显示区域小于它包含的内容所需要的显示“面积”——术语称这种情况为“内容溢出”，浏览器将根据这个元素的 CSS overflow 属性来显示那些内容。在发生内容溢出时，浏览器将根据 overflow 属性的值对内容进行“裁剪”，其效果是内容只有一部分在浏览器窗口里是可见的。我们还可以通过 overflow 属性告诉浏览器是否需要显示滚动条，以便让用户能够看到内容的其他部分。

overflow 属性的可取值有 4 种：visible、hidden、scroll 和 auto。它们的含义是：

- visible：不裁剪溢出的内容。浏览器将把溢出的内容呈现在其容器元素的显示区域以外的地方，全部内容在浏览器窗口里都是可见的。
- hidden：裁剪溢出的内容。内容只显示在其容器元素的显示区域里，这意味着只有一部分内容在浏览器窗口里是可见的。
- scroll：类似于 hidden，浏览器将对溢出的内容进行裁剪，但会显示滚动条以便让用户能够看到内容的其他部分。
- auto：类似于 scroll，但浏览器只在真的发生内容溢出时才显示滚动条。如果内容没有溢出，就不显示滚动条。

如此说来，在 overflow 属性的 4 种可取值当中，最能满足我要求的显然是 hidden。我有一张实际尺寸是 400×100 像素的图片，但我每次只想显示这张图片中一个尺寸为 100 像素×100

像素的部分。

首先，需要把这张图片放到一个容器元素里。我将把它放入一个 div 元素，并把这个 div 元素的 id 属性值设置为 slideshow：

```
<div id="slideshow">
 <img src="topics.gif" alt="building blocks of web design"
➥  id="preview" />
</div>
```

现在，将创建一个样式表文件 layout.css，然后把它放入 styles 文件夹里。

在 layout.css 文件里，我对 id="slideshow"的 div 元素的尺寸做了如下设置：

```
#slideshow {
  width: 100px;
  height: 100px;
  position: relative;
}
```

为了确保这个 div 元素里的内容将被裁剪，我还需要把它的 CSS overflow 属性设置为 hidden：

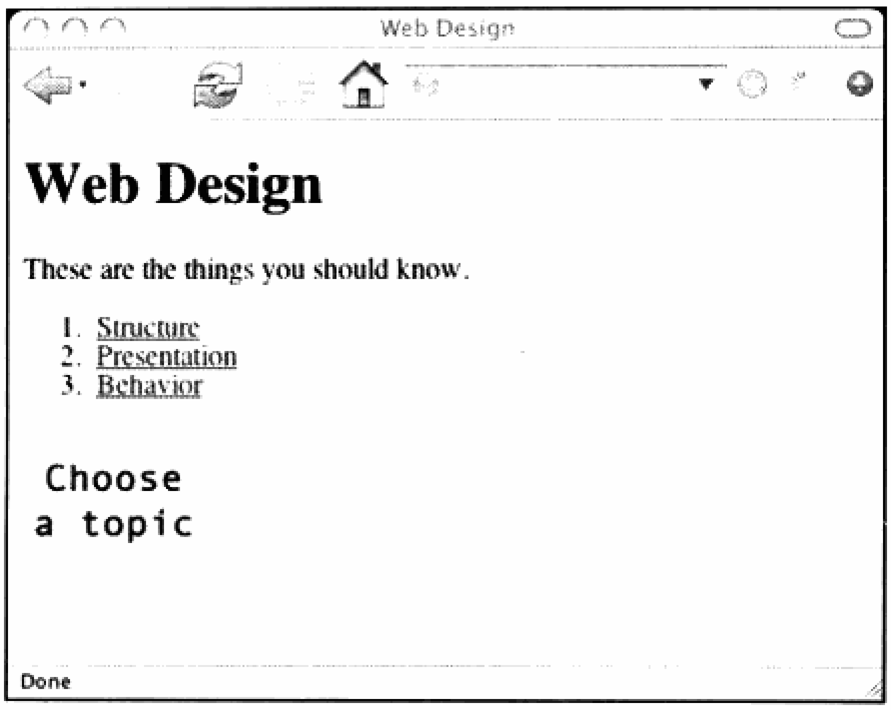
```
#slideshow {
  width: 100px;
  height: 100px;
  position: relative;
  overflow: hidden;
}
```

接下来，用 CSS 提供的@import 命令把 layout.css 样式表引入 list.html 文档，它被包含到文档头部的<style>标签里：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Web Design</title>
  <style type="text/css" media="screen">
    @import url("styles/layout.css");
  </style>
</head>
<body>
  <h1>Web Design</h1>
  <p>These are the things you should know.</p>
  <ol id="linklist">
    <li>
      <a href="structure.html">Structure</a>
    </li>
    <li>
      <a href="presentation.html">Presentation</a>
```

```
      </li>
      <li>
        <a href="behavior.html">Behavior</a>
      </li>
    </ol>
    <div id="slideshow">
      <img src="topics.gif" alt="building blocks of web design"
➥ id="preview" />
    </div>
</body>
</html>
```

现在，把 list.html 文档加载到一个 Web 浏览器，就可以看到相应的变化（图片已经被裁剪了）。



现在，我们在浏览器窗口里只能看到 topics.gif 图片的一部分——它只有第一个 100 像素宽的部分是可见的。

我接下来要解决的问题是，让这个网页对用户的操作行为做出正确的响应。我想在用户把鼠标指针悬停在某个链接上时，把 topics.gif 图片中与之对应的那个部分显示出来。这是一种行为上的变化：用 JavaScript 和 DOM 来实现是再合适不过了。

10.2.4　JavaScript 代码

我的计划是用 moveElement()函数来移动 topics.gif 图片。根据用户正把鼠标指针悬停在哪个链接上，我将把这个图片向左或向右移动。

我需要把调用 moveElement()函数的操作行为与链接清单里每个链接的 onmouseover 事件关联起来。

我编写了一个 prepareSlideshow()函数来完成这项工作，下面是它的代码清单：

```
function prepareSlideshow() {
// Make sure the browser understands the DOM methods
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
// Make sure the elements exist
  if (!document.getElementById("linklist")) return false;
  if (!document.getElementById("preview")) return false;
// Apply styles to the preview image
  var preview = document.getElementById("preview");
  preview.style.position = "absolute";
  preview.style.left = "0px";
  preview.style.top = "0px";
// Get all the links in the list
  var list = document.getElementById("linklist");
  var links = list.getElementsByTagName("a");
// Attach the animation behavior to the mouseover event
  links[0].onmouseover = function() {
    moveElement("preview",-100,0,10);
  }
  links[1].onmouseover = function() {
    moveElement("preview",-200,0,10);
  }
  links[2].onmouseover = function() {
    moveElement("preview",-300,0,10);
  }
}
```

首先，prepareSlideshow()函数要检查浏览器能否理解它将用到的 DOM 方法：

```
if (!document.getElementsByTagName) return false;
if (!document.getElementById) return false;
```

接着，检查 linklist 和 preview 元素是否真的存在。别忘了，preview 是 topics.gif 图片的 id 属性值：

```
if (!document.getElementById("linklist")) return false;
if (!document.getElementById("preview")) return false;
```

此后，为“preview”图片设定一个默认位置。我将把它的 style.left 属性设置为 0px，把它的 style.top 属性也设置为 0px：

```
var preview = document.getElementById("preview");
preview.style.position = "absolute";
preview.style.left = "0px";
preview.style.top = "0px";
```

请注意，这并不意味着 topics.gif 图片将出现在浏览器窗口的左上角。它将出现在它的容器元素——也就是那个 id 属性值是 slideshow 的 div 元素的左上角。因为那个 div 元素的 CSS position 属性值是 relative：如果把 position 属性值是 absolute 的元素放入一个 position 属

性值是 relative 的元素，后者就成为了前者的容器元素，而前者将在后者的显示区域里按 absolute 方式进行摆放。换句话说，preview 图片将出现在 slideshow 元素的左上角——与这个 div 元素的左边界和上边界之间的距离都是 0px。

最后，把 onmouseover 行为与链接清单里的各个链接关联起来。首先，把一个由包容在 linklist 元素里的所有 a 元素构成的节点集合赋值给变量 links。第一个链接对应着 links[0]，第二个链接对应着 links[1]，第三个链接对应着 links[2]：

```
var list = document.getElementById("linklist");
var links = list.getElementsByTagName("a");
```

当用户把鼠标指针悬停在第一个链接上时，moveElement()函数将被调用执行。此时，它的 elementID 参数的值是 preview，final_x 参数的值是-100，final_y 参数的值是 0，interval 参数的值是 10 毫秒：

```
links[0].onmouseover = function() {
  moveElement("preview",-100,0,10);
}
```

第二个链接应该有同样的行为——除了 final_x 参数的值变成了-200：

```
links[1].onmouseover = function() {
  moveElement("preview",-200,0,10);
}
```

第三个链接将把 preview 图片向左移动-300 像素：

```
links[2].onmouseover = function() {
  moveElement("preview",-300,0,10);
}
```

接下来，用 addLoadEvent()函数调用 prepareSlideshow()函数，这将使得后者在页面加载时得到执行并把 onmouseover 行为绑定到那三个链接上：

```
addLoadEvent(prepareSlideshow);
```

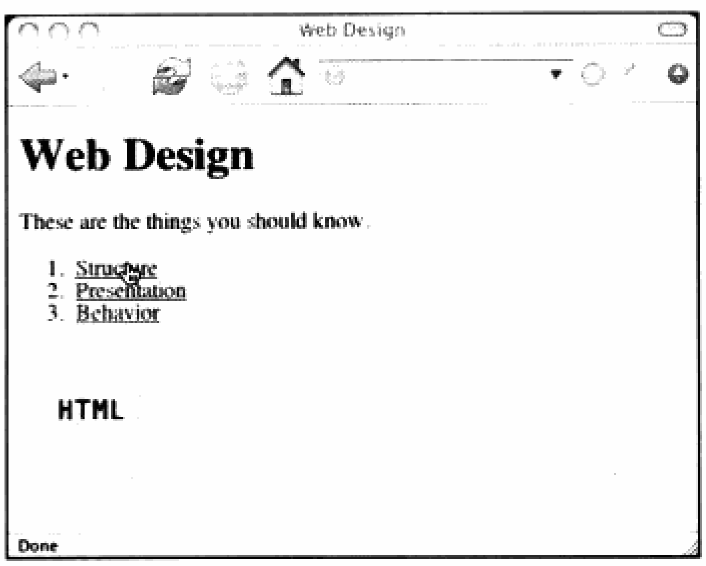
把 prepareSlideshow()函数保存为 prepareSlideshow.js 文件并将其放到 scripts 文件夹里。把 moveElement.js 和 addLoadEvent.js 文件也放到同一个文件夹中。

为了从 list.html 文档里调用这三个脚本，我还需要在这个文档的<head>部分添加一些<script>标签：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Web Design</title>
  <style type="text/css" media="screen">
    @import url("styles/layout.css");
```

```
  </style>
  <script type="text/javascript" src="scripts/addLoadEvent.js">
  </script>
  <script type="text/javascript" src="scripts/moveElement.js">
  </script>
  <script type="text/javascript" src="scripts/prepareSlideshow.js">
  </script>
</head>
<body>
  <h1>Web Design</h1>
  <p>These are the things you should know.</p>
  <ol id="linklist">
    <li>
      <a href="structure.html">Structure</a>
    </li>
    <li>
      <a href="presentation.html">Presentation</a>
    </li>
    <li>
      <a href="behavior.html">Behavior</a>
    </li>
  </ol>
  <div id="slideshow">
    <img src="topics.gif" alt="building blocks of web design"
➥ id="preview" />
  </div>
</body>
</html>
```

把 list.html 文档加载到一个 Web 浏览器。把鼠标指针悬停在清单里的某个链接上就可以看到动画效果。



根据鼠标指针正悬停在哪个链接上，topics.gif 图片的不同部分将进入我们的视线。

不过，有些事情好像不太对头：如果你把鼠标指针在链接之间快速地来回移动，动画效果将

变得混乱起来。这说明 moveElement()函数的什么地方有问题。

10.2.5　与变量的作用域有关的问题

动画效果不正确的问题是由一个全局变量引起的。

在把 moveMessage()函数抽象化为 moveElement()函数的过程中，我未对变量 movement 做任何修改：

```
function moveElement(elementID,final_x,final_y,interval) {
  if (!document.getElementById) return false;
  if (!document.getElementById(elementID)) return false;
  var elem = document.getElementById(elementID);
  var xpos = parseInt(elem.style.left);
  var ypos = parseInt(elem.style.top);
  if (xpos == final_x && ypos == final_y) {
    return true;
  }
  if (xpos < final_x) {
    xpos++;
  }
  if (xpos > final_x) {
    xpos--;
  }
  if (ypos < final_y) {
    ypos++;
  }
  if (ypos > final_y) {
    ypos--;
  }
  elem.style.left = xpos + "px";
  elem.style.top = ypos + "px";
  var repeat =
➥ "moveElement('"+elementID+"',"+final_x+","+final_y+","+interval+")";
  movement = setTimeout(repeat,interval);
}
```

这留下了一个隐患：每当用户把鼠标指针悬停在某个链接上，不管上一次调用是否已经把图片移动到位，moveElement()函数都会被再次调用并试图把同一个元素移动到另一个地方去。于是，当用户在链接之间快速移动鼠标时，movement 变量就会像一条拔河绳那样来回变化，而 moveElement()函数就会试图把同一个元素同时移动到两个不同的地方去。如果用户移动鼠标的速度够快，积累在 setTimeout 队列里的事件就会导致动画效果产生滞后。

为了消除动画滞后的现象，我可以用 clearTimeout()函数清除积累在 setTimeout 队列里的事件：

```
clearTimeout(movement);
```

可是，如果在还没有设置 movement 变量之前就执行这条语句，我将“制造”出一个错误。

我不能使用一个局部变量：

```
var movement = setTimeout(repeat,interval);
```

如果我那样做了，clearTimeout()函数调用语句将无法工作——因为局部变量 movement 在 clearTimeout()函数的上下文里不存在。

也就是说，我既不能使用一个全局变量，也不能使用一个局部变量。我需要一种介乎它们二者之间的东西，我需要一个只与正在被移动的那个元素有关的变量。

只与某个特定元素有关的变量是存在的。事实上，我们一直在使用它们。那就是“属性”。

到目前为止，我们一直在使用由 DOM 提供的属性，如 *element.firstChild*、*element.style*，等等。

JavaScript 允许我们为元素创建属性：

```
element.property = value
```

比如说，只要愿意，我们完全可以创建一个名为 foo 的属性并把它设置为"bar"：

```
element.foo = "bar";
```

这很像是在创建一个变量，但区别是这个变量专属于某个特定的元素。

我将把变量 movement 从一个全局变量改变为正在被移动的那个元素（elem 元素）的属性。这样一来，就可以测试它是否已经存在，并在它已经存在的情况下使用 clearTimeout()函数了：

```
function moveElement(elementID,final_x,final_y,interval) {
  if (!document.getElementById) return false;
  if (!document.getElementById(elementID)) return false;
  var elem = document.getElementById(elementID);
  if (elem.movement) {
    clearTimeout(elem.movement);
  }
  var xpos = parseInt(elem.style.left);
  var ypos = parseInt(elem.style.top);

  if (xpos == final_x && ypos == final_y) {
    return true;
  }
  if (xpos < final_x) {
    xpos++;
  }
  if (xpos > final_x) {
    xpos--;
  }
  if (ypos < final_y) {
    ypos++;
```

```
    }
    if (ypos > final_y) {
      ypos--;
    }
    elem.style.left = xpos + "px";
    elem.style.top = ypos + "px";
    var repeat =
➥ "moveElement('"+elementID+"',"+final_x+","+final_y+","+interval+")";
    elem.movement = setTimeout(repeat,interval);
  }
```

于是，不管 moveElement()函数正在移动的是哪个元素，该元素都将获得一个名为 movement 的属性。如果该元素在 moveElement()函数开始执行时已经有了一个 movement 属性，就应该用 clearTimeout()函数对它进行复位。这样一来，即使因为用户快速移动鼠标指针而使得某个元素需要向不同的方向移动，实际执行的也只有一条 setTimeout()函数调用语句。

请重新加载 list.html 文件。现在，在链接之间快速移动鼠标指针将不再引起任何问题。setTimeout 队列里不再有积累的事件，动画将随着鼠标指针在链接之间的移动而立刻改变方向。

接下来，再来看看我们还可以对动画效果做哪些改进。

10.3 改进动画效果

在元素到达由 final_x 和 final_y 参数给出的目的地之前，moveElement()函数每次只把它移动一个像素（1px）的距离。移动效果很平滑，但移动速度未免有些慢。

我将把动画的移动速度加快一点儿。

请仔细看看下面这些简单的代码，它们来自 moveElement.js 文件：

```
if (xpos < final_x) {
  xpos++;
}
```

变量 xpos 是被移动元素的当前左位置，变量 final_x 是这个元素的目的地的左位置。上面这段代码的含义是："如果变量 xpos 小于变量 final_x，就给 xpos 的值加 1。"

也就是说，不管那个元素与它的目的地距离多远，它每次只前进一个像素（1px）。我想改变这种慢悠悠的步调：如果那个元素与它的目的地距离较远，就让它每次前进一大步；如果那个元素与它的目的地距离较近，就让它每次前进一小步。

首先，我需要算出元素与它的目的地之间的距离。如果 xpos 小于 final_x，我想知道它们差多少。只要用 final_x（目的地的左位置）减去 xpos（当前左位置）就可以知道答案：

```
var dist = final_x - xpos;
```

这个结果就是元素还需要行进的距离。我决定让元素前进这个距离的十分之一。我选用十分之一

的理由是为了计算方便；如果你们愿意，选用其他的值也没问题：

```
var dist = (final_x - xpos)/10;
xpos = xpos + dist;
```

这将把元素朝它的目的地移动十分之一的距离。

如果 xpos 与 final_x 相差 500 像素，变量 dist 将等于 50。xpos 的值将增加 50。

如果 xpos 与 final_x 相差 100 像素，xpos 的值将增加 10。

不过，当 xpos 与 final_x 之间的差距小于 10 的时候，问题来了：用这个差距除以 10 的结果将小于 1，而我们不可能把一个元素移动不到一个像素的距离。

这个问题可以用 Math 对象的 ceil 属性来解决，它可以把变量 dist 的值向“大于”方向舍入为一个整数。

下面是 ceil 属性的语法：

```
Math.ceil(number)
```

这将把浮点数 *number* 向“大于”方向舍入为与之最接近的整数。还有一个与此相对的 floor 属性，它可以把任意浮点数向“小于”方向舍入为与之最接近的整数。round 属性将把任意浮点数舍入为与之最接近的整数：

```
Math.floor(number)
Math.round(number)
```

具体到 moveElement()函数，我需要向“大于”方向进行舍入。如果错误地选用了 floor 或 round 属性，那个元素将永远也不会到达它的目的地：

```
var dist = Match.ceil((final_x - xpos)/10);
xpos = xpos + dist;
```

这就解决了 xpos 小于 final_x 时的问题：

```
if (xpos < final_x) {
  var dist = Math.ceil((final_x - xpos)/10);
  xpos = xpos + dist;
}
```

如果 xpos 大于 final_x，在计算距离时就应该用 xpos 减去 final_x。把这个减法结果除以 10，向“大于” 舍入为与之最接近的整数,然后赋值给变量 dist。此时，我们必须用 xpos 减去 dist 才能让元素更接近它的目的地：

```
if (xpos > final_x) {
  var dist = Math.ceil((xpos - final_x)/10);
  xpos = xpos - dist;
}
```

同样的逻辑也适用于变量 ypos 和 final_y：

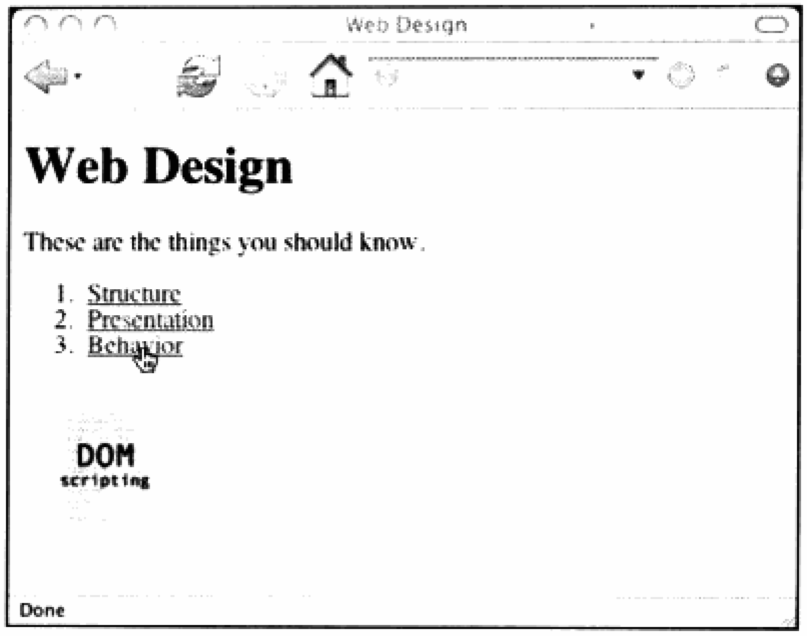
```
if (ypos < final_y) {
  var dist = Math.ceil((final_y - ypos)/10);
  ypos = ypos + dist;
}
if (ypos > final_y) {
  var dist = Math.ceil((ypos - final_y)/10);
  ypos = ypos - dist;
}
```

下面是 moveElement()函数在经过上述改进后的代码清单：

```
function moveElement(elementID,final_x,final_y,interval) {
  if (!document.getElementById) return false;
  if (!document.getElementById(elementID)) return false;
  var elem = document.getElementById(elementID);
  if (elem.movement) {
    clearTimeout(elem.movement);
  }
  var xpos = parseInt(elem.style.left);
  var ypos = parseInt(elem.style.top);
  if (xpos == final_x && ypos == final_y) {
    return true;
  }
  if (xpos < final_x) {
    var dist = Math.ceil((final_x - xpos)/10);
    xpos = xpos + dist;
  }
  if (xpos > final_x) {
    var dist = Math.ceil((xpos - final_x)/10);
    xpos = xpos - dist;
  }
  if (ypos < final_y) {
    var dist = Math.ceil((final_y - ypos)/10);
    ypos = ypos + dist;
  }
  if (ypos > final_y) {
    var dist = Math.ceil((ypos - final_y)/10);
    ypos = ypos - dist;
  }
  elem.style.left = xpos + "px";
  elem.style.top = ypos + "px";
  var repeat =
➥ "moveElement('"+elementID+"',"+final_x+","+final_y+","+interval+")";
  elem.movement = setTimeout(repeat,interval);
}
```

把这些修改保存到 moveElement.js 文件。重新加载 list.html 就可以看到新的动画效果：

现在，动画效果给人的感觉是更加平滑和迅速。当你第一次把鼠标指针悬停在某个链接上时，图片将跳跃一大段距离。随着图片越来越接近它的最终目的地，它会“放慢”自己的脚步。



在(X)HTML、CSS 和 JavaScript 的共同努力下，预期的动画效果终于实现了。一切都显得那么完美，但凡事都有改进的余地，这一次也不例外。

10.4 最后的优化

moveElement()函数现在的表现确实非常好，但还有一件事让我感到不放心：这个函数的开头部分需要依赖于一个假设：

```
var xpos = parseInt(elem.style.left);
var ypos = parseInt(elem.style.top);
```

看出来了吗？这里需要假设 elem 元素肯定有一个 left 样式属性和一个 top 样式属性。我其实应该先检查一下这是不是事实。

如果 elem 元素的 left 和/或 top 属性未被设置，我有以下几种选择。首先，可以简单地就此退出这个函数：

```
if (!elem.style.left || !elem.style.top) {
  return false;
}
```

如果 JavaScript 没有读到这些属性，整个函数将静悄悄地结束运行而不是报告出错。

另一种选择是在 moveElement()函数里为 left 和 top 属性分别设置一个默认值：如果这两个属性没有被设置，我将把它们的默认值设置为 0px：

```
if (!elem.style.left) {
  elem.style.left = "0px";
}
if (!elem.style.top) {
  elem.style.top = "0px";
}
```

下面是 moveElement()函数现在的代码清单：

```
function moveElement(elementID,final_x,final_y,interval) {
  if (!document.getElementById) return false;
  if (!document.getElementById(elementID)) return false;
  var elem = document.getElementById(elementID);
  if (elem.movement) {
    clearTimeout(elem.movement);
  }
  if (!elem.style.left) {
    elem.style.left = "0px";
  }
  if (!elem.style.top) {
    elem.style.top = "0px";
  }
  var xpos = parseInt(elem.style.left);
  var ypos = parseInt(elem.style.top);
  if (xpos == final_x && ypos == final_y) {
    return true;
  }
  if (xpos < final_x) {
    var dist = Math.ceil((final_x - xpos)/10);
    xpos = xpos + dist;
  }
  if (xpos > final_x) {
    var dist = Math.ceil((xpos - final_x)/10);
    xpos = xpos - dist;
  }
  if (ypos < final_y) {
    var dist = Math.ceil((final_y - ypos)/10);
    ypos = ypos + dist;
  }
  if (ypos > final_y) {
    var dist = Math.ceil((ypos - final_y)/10);
    ypos = ypos - dist;
  }
  elem.style.left = xpos + "px";
  elem.style.top = ypos + "px";
  var repeat =
➥ "moveElement('"+elementID+"',"+final_x+","+final_y+","+interval+")";
  elem.movement = setTimeout(repeat,interval);
}
```

有了刚才所说的保险措施之后，就用不着再明确地设置 preview 元素的出发点位置了。这意味着可以把 prepareSlideshow()函数里如下所示的两条现有语句删掉：

```
preview.style.left = "0px";
preview.style.top = "0px";
```

既然提到了 prepareSlideshow()函数，我决定仔细看看它是不是还有其他地方需要改进。

生成 HTML 内容

在 list.html 文档里包含一些只是为了能够用 JavaScript 代码实现动画效果而存在的 HTML 代码：

```
<div id="slideshow">
  <img src="topics.gif" alt="building blocks of web design"
➥ id="preview" />
</div>
```

如果用户没有激活 JavaScript 支持功能，以上内容就未免有些多余了。这里的 div 和 img 元素纯粹是为了动画效果才“塞”进来的。既然如此，与其把这些元素硬编码在文档里，不如用 JavaScript 代码来生成它们。我决定在 prepareSlideshow.js 文件里做这些事情。

首先，创建 div 元素：

```
var slideshow = document.createElement("div");
slideshow.setAttribute("id","slideshow");
```

接着，创建 img 元素：

```
var preview = document.createElement("img");
preview.setAttribute("src","topics.gif");
preview.setAttribute("alt","building blocks of web design");
preview.setAttribute("id","preview");
```

把新创建的 img 元素放入新创建的 div 元素：

```
slideshow.appendChild(preview);
```

最后，我想让这些新创建的元素紧跟着出现在链接清单的后面。我将使用来自本书第 7 章的 insertAfter()函数来完成这一步骤：

```
var list = document.getElementById("linklist");
insertAfter(slideshow,list);
```

下面是最终完成的 prepareSlideshow()函数的代码清单：

```
function prepareSlideshow() {
// Make sure the browser understands the DOM methods
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
// Make sure the elements exist
  if (!document.getElementById("linklist")) return false;
  var slideshow = document.createElement("div");
  slideshow.setAttribute("id","slideshow");
  var preview = document.createElement("img");
  preview.setAttribute("src","topics.gif");
  preview.setAttribute("alt","building blocks of web design");
  preview.setAttribute("id","preview");
  slideshow.appendChild(preview);
```

```
    var list = document.getElementById("linklist");
    insertAfter(slideshow,list);
  // Get all the links in the list
    var links = list.getElementsByTagName("a");
  // Attach the animation behavior to the mouseover event
    links[0].onmouseover = function() {
      moveElement("preview",-100,0,10);
    }
    links[1].onmouseover = function() {
      moveElement("preview",-200,0,10);
    }
    links[2].onmouseover = function() {
      moveElement("preview",-300,0,10);
    }
  }
addLoadEvent(prepareSlideshow);
```

接下来，还需要对 list.html 文件做一些修改：删除 id="slideshow"的 div 元素和 id="preview"的图片；添加一组<script>标签来调用 insertAfter.js 文件。下面是最终完成的 list.html 文件的代码清单：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Web Design</title>
  <style type="text/css" media="screen">
    @import url(styles/layout.css);
  </style>
  <script type="text/javascript" src="scripts/addLoadEvent.js">
  </script>
  <script type="text/javascript" src="scripts/insertAfter.js">
  </script>
  <script type="text/javascript" src="scripts/moveElement.js">
  </script>
  <script type="text/javascript" src="scripts/prepareSlideshow.js">
  </script>
</head>
<body>
  <h1>Web Design</h1>
  <p>These are the things you should know.</p>
  <ol id="linklist">
    <li>
      <a href="structure.html">Structure</a>
    </li>
    <li>
      <a href="presentation.html">Presentation</a>
    </li>
```

```
    <li>
      <a href="behavior.html">Behavior</a>
    </li>
  </ol>
</body>
</html>
```

把 insertAfter()函数写入 insertAfter.js 文件并把它放入 scripts 文件夹：

```
function insertAfter(newElement,targetElement) {
  var parent = targetElement.parentNode;
  if (parent.lastChild == targetElement) {
    parent.appendChild(newElement);
  } else {
    parent.insertBefore(newElement,targetElement.nextSibling);
  }
}
```

我还需要对样式表文件 layout.css 做一些修改。因为我刚才从 prepareSlideshow.js 文件里删除了如下所示的一行代码：

```
preview.style.position = "absolute";
```

所以现在需要把以下样式声明添加到 layout.css 样式表里，这才是样式信息应该属于的地方：

```
#slideshow {
  width: 100px;
  height: 100px;
  position: relative;
  overflow: hidden;
}
#preview {
  position: absolute;
}
```

现在，在 Web 浏览器里刷新 list.html 文档。从表面上看，功能还是那些功能，行为还是那些行为。但经过上述改进之后，这份文档的结构层、表示层和行为层已经分离得更加彻底了。如果在禁用了 JavaScript 支持功能的情况下浏览这份文档，动画图片将根本不会出现。

我们用 JavaScript 实现的动画功能非常完善。如果激活了 JavaScript，这个页面就能根据用户的操作动作通过动画效果向用户提供一些赏心悦目的视觉反馈；如果没有激活 JavaScript，动画功能将按照我们安排的预留退路保持静默，不影响用户的浏览体验。

如果想进一步加强链接清单和动画图片的视觉联系，可以通过编辑 layout.cs 文件的办法去实现一些更精彩的效果。比如说，可以把动画图片的显示位置从链接清单的下方挪到它的旁边。如果想让动画部分更加突出的话，还可以给它加上一个边框。

10.5 小结

在这一章里，我们首先对“动画”进行了定义：随时间变化而改变某个元素在浏览器窗口里的显示位置。然后，我们令人信服地证明了这样一个结论：通过结合使用 CSS-DOM 和 JavaScript 的 setTimeout()函数，创建一个简单的动画并不困难。

从技术上讲，实现动画效果并不困难，问题是在实践中应不应该使用动画。动画技术可以让我们创建出很多种非常酷的效果，但那些四处移动的元素对用户有用或有帮助的场合却并不多。不过，我们刚才创建的 JavaScript 动画却是一个与众不同的例外。我们花了不少功夫才让它有了平滑的动画效果和足够的预留退路，而最终的结果证明我们付出的努力是非常值得的。

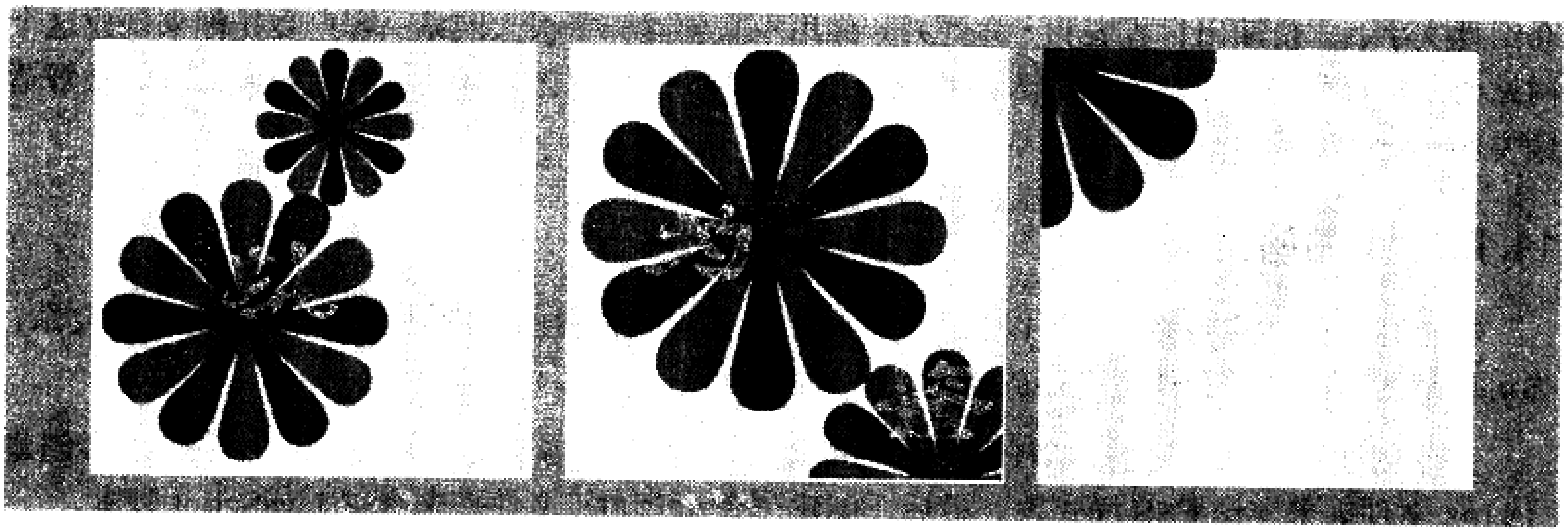
我们现在有了一个通用性的函数，它可以在确有必要创建一种这样或那样的动画效果时帮上我们的大忙。

在 JavaScript 语言和 DOM 的帮助下，我们可以从许多方面对网页的内容和呈现效果进行充实和完善，用 JavaScript 代码来实现动画效果只是其中的一种。到目前为止，我们已经在这本书里向大家分门别类地介绍了很多种用 DOM 脚本去充实网页的思路和技巧。在下一章里，请大家和我一起把已经学到的所有思路和技巧结合起来运用到实践中。

所谓学以致用，学了那么多的技巧，也该用它们去解决一些实际的问题了。

第 11 章

学以致用：JavaScript 网站设计实战



本章内容

- ❑ 编排内容
- ❑ 设计网站
- ❑ 施用样式
- ❑ 用 JavaScript 和 DOM 完善网站

在本书此前的章节里，你们已经见过许多 DOM 的应用示例了，但那些示例彼此没有联系，而且比较偏重于理论。现在是把它们结合起来去解决一个实际问题的时候了。在这一章里，我们将从零开始去创建一个网站并用 JavaScript 来完善它。

11.1 案例背景介绍

你是一位幸运的网站设计师，因为你被选中为这个星球上最酷的乐队之一"Jay Skript and the Domsters"设计一个网站！

好吧，我承认这个乐队并不存在。但我希望你能接受这个任务。为了完成这一章的学习，我们不仅需要假设这个乐队肯定存在，还需要假设确实有人请你去为这个乐队设计一个网站。

这个网站应该像这支乐队那样给人以一种很酷的感觉。因此，如果你能给这个网站增加一些与众不同的交互功能的话，那将会非常好。不过，这个网站还需要有良好的可访问性并支持各大搜索引擎的搜索。

这个网站的用途是向大众提供关于这个乐队的信息。不管你做出什么样的设计决定，都必须把这一点放在首位。

这就是你的任务。

11.1.1　原始材料

你的客户已经提供了一些供你搭建这个网站的基本素材。你手里现在有一些关于这个乐队的介绍性文字、一份演出日程表和一些照片。你用不着创建一个很大网站。客户委托你来创建这个网站的基本目的是为了让更多的人熟悉这个乐队，所以你必须让来到这个网站的人们有一种舒适愉悦的感觉。

11.1.2　网站的结构

根据客户提供的内容，你可以相当于轻松地创建出一份网站地图。它的结构并不是很复杂，你可以把所有的页面文件集中存放在同一个文件夹里。

开始创建这个网站之前，请创建一个 `images` 文件夹来容纳你将用到的图片文件。再创建 `styles` 文件夹来容纳 CSS 样式表。最后，创建一个 `scripts` 文件夹来容纳 JavaScript 脚本。

你的子目录结构现在应该是如下所示的样子：

- /images
- /styles
- /scripts

你需要通过页面来提供关于这个乐队的所有背景信息。你可以把那些照片集中放在另一个页面上的图片库里。乐队的演出日程又需要有一个自己的页面。你还需要创建一个“联系”页面好让访问者能够与这个乐队进行交流。最后，这个网站还需要有一个对其他页面进行简单介绍的主页，访问者将从这里开始他们在这个网站上的旅程。

综上所述，你将需要编写以下几个网页，如图 11-1 所示：

- Home：这个网站的主页
- About：乐队简介
- Photos：乐队的演出照片
- Live：乐队的演出日程
- Contact：供访问者与乐队进行交流

你的具体任务是把上面这份清单转换为以下这些文件：

- index.html
- about.html
- photos.html
- live.html
- contact.html

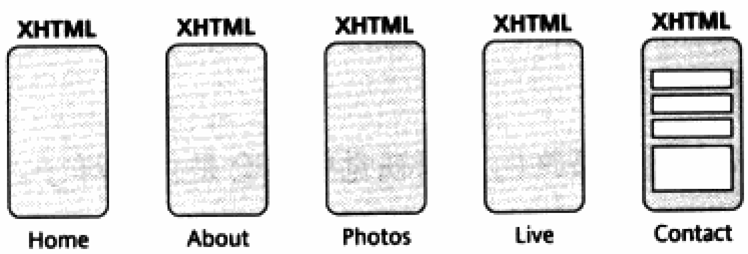


图 11-1 “Jay Skript and the Domsters”乐队网站的网站地图

虽然这些页面的内容会各不相同，但它们应该有同样的基本结构。因此，应该先为这些页面创建一个通用的模板。

11.1.3 网页的结构

这个网站的每一个页面可以划分为以下几大部分：

(1) 标题部分：相当于这个网站的品牌，这里是网站 Logo 标志的最佳去处。

(2) 导航部分：你将把通往各个页面的链接排列在这里。

(3) 内容部分：这是每个页面的精华所在。

考虑到要把这些页面划分为几大部分，使用<div>标签就是一件顺理成章的事情了。这些<div>标签的 id 属性值将分别是 header、navigation 和 content。

你可以轻车熟路地创建一个如下所示的通用模板：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Jay Skript And The Domsters</title>
</head>
<body>
  <div id="header">
  </div>
  <div id="navigation">
    <ul>
      <li><a href="index.html">Home</a></li>
      <li><a href="about.html">About</a></li>
```

```
      <li><a href="photos.html">Photos</a></li>
      <li><a href="live.html">Live</a></li>
      <li><a href="contact.html">Contact</a></li>
    </ul>
  </div>
  <div id="content">
  </div>
</body>
</html>
```

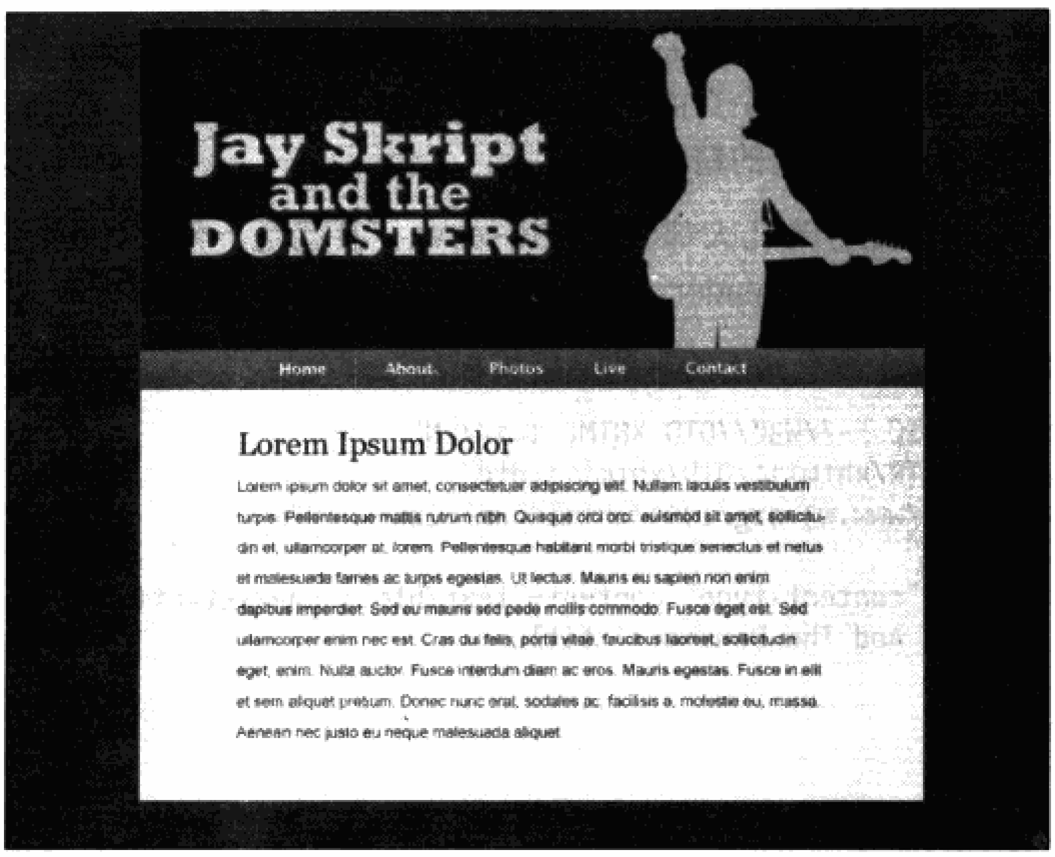
把以上代码保存为 template.html 文件。现在已经有了一个基本的结构，接下来的工作是把有关内容分别插入到各个页面里去。

在那之前，我们不妨先去看看最终的页面视觉效果会是什么样子。

11.2 页面视觉效果设计

现在，已经知道这个网站的每个页面都将包含哪些结构性元素了。掌握了这些情况，再加上客户提供的原始素材，你就可以着手设计那些页面的视觉效果了。你可以选用 Photoshop、Firework 或其他图形设计工具来完成这项工作，在这方面你有完全的自由。

我当然猜不到你设计出来的视觉效果是怎样的，但好在这个问题并不是我们这里关心的焦点。虽然本章的主角是你，但我现在要喧宾夺主一下——下面是我刚完成的设计。



在把页面的视觉效果最终确定下来之后，你将需要把整个画面分解为一些图形元素：把整个页面的背景图片保存为 background.gif 文件，把乐队名称图片保存为 logo.gif 文件，把导航条

的背景图片保存为 navbar.gif 文件，把那个振臂高呼的人物剪影保存为 guitarist.gif 文件。最后，把这些文件统一存放到 images 文件夹里去。

11.3 CSS

现在，已经有了一个基本的 XHTML 模板，也已经设计好了这个网站的视觉效果。为了让网页呈现出这样的效果，你需要为你的模板声明一些 CSS。你需要编写一些 CSS 文件。

你可以把所有的 CSS 信息写在同一个文件里，但那会给以后的 CSS 编辑工作带来很多不便。如果从一开始就把你的 CSS 信息分门别类地保存在多个文件里的话，以后修改起来将会很方便。

如何分散保存 CSS 信息是你的自由，但我建议你把那些与页面布局有关的 CSS 信息保存到一个文件里，把与颜色有关的 CSS 信息保存到另一个文件里，再用一个文件来专门保存与文本字型有关的 CSS 信息：如下所示：

- layout.css
- color.css
- typography.css

这些文件可以用如下所示的代码引入一个基本样式表：

```
@import url(layout.css);
@import url(color.css);
@import url(typography.css);
```

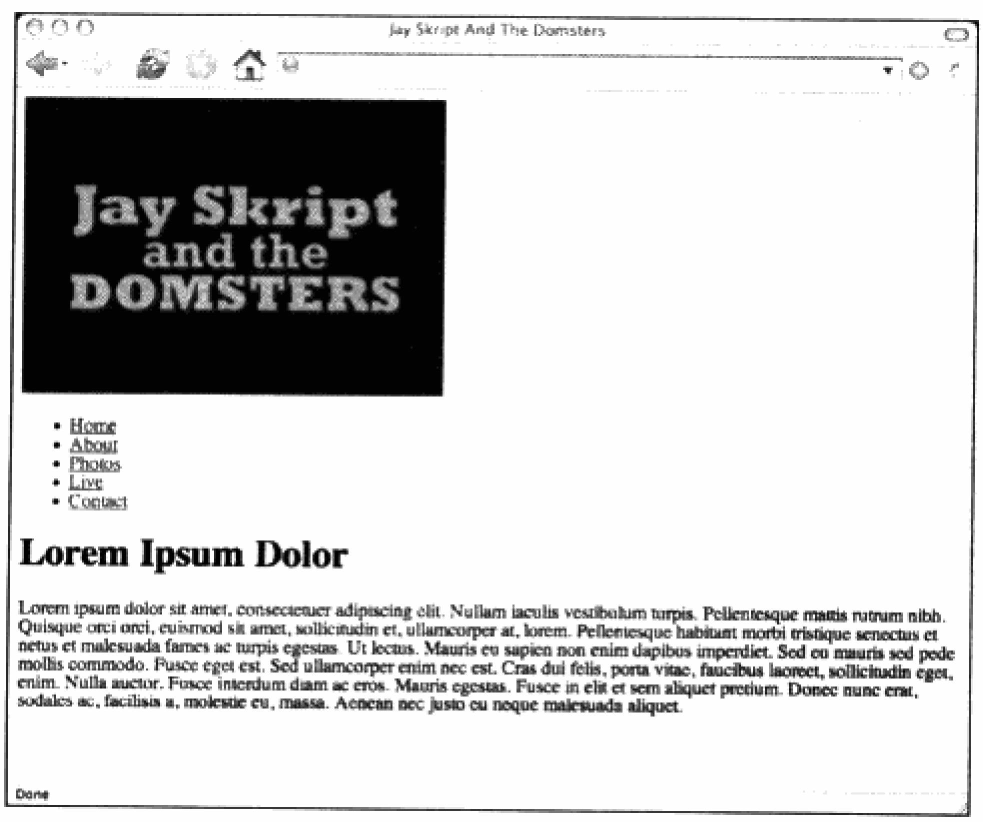
把这三行代码保存为 styles 文件夹里的 basic.css 文件。如果今后需要增加一个新的样式表或者是需要删除一个现有的样式表，你只需对 basic.css 文件进行编辑。

为了让页面模板能够调用这个基本样式表，你需要在那份文档的<head>部分插入一个<link>标签。趁着这个机会，你还可以用一个<img>标签把网站 Logo 图片添加到 id 属性值等于 header 的那个 div 元素里去。为便于看出网页的呈现效果，还可以在这份文档里添加一些示例内容，如“lorem ipsum ...”部分所示：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Jay Skript And The Domsters</title>
  <link rel="stylesheet" type="text/css" media="screen"
➥ href="styles/basic.css" />
</head>
<body>
  <div id="header">
    <img src="images/logo.gif" alt="Jay Skript and the Domsters" />
  </div>
  <div id="navigation">
```

```
      <ul>
        <li><a href="index.html">Home</a></li>
        <li><a href="about.html">About</a></li>
        <li><a href="photos.html">Photos</a></li>
        <li><a href="live.html">Live</a></li>
        <li><a href="contact.html">Contact</a></li>
      </ul>
    </div>
    <div id="content">
      <h1>Lorem Ipsum Dolor</h1>
      <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
  Nullam iaculis vestibulum turpis. Pellentesque mattis rutrum
  nibh. Quisque orci orci, euismod sit amet, sollicitudin et,
  ullamcorper at, lorem.
  Pellentesque habitant morbi tristique senectus et netus
  et malesuada fames ac turpis egestas.
  Ut lectus. Mauris eu sapien non enim dapibus imperdiet.
  Sed eu mauris sed pede mollis commodo.
  Fusce eget est. Sed ullamcorper enim nec est.
  Cras dui felis, porta vitae, faucibus laoreet, sollicitudin eget,
  enim. Nulla auctor. Fusce interdum diam ac eros.
  Mauris egestas. Fusce in elit et sem aliquet pretium.
  Donec nunc erat, sodales ac, facilisis a, molestie eu, massa.
  Aenean nec justo eu neque malesuada aliquet.</p>
    </div>
  </body>
  </html>
```

下面是模板文档 template.html 在未使用任何样式表时的样子。



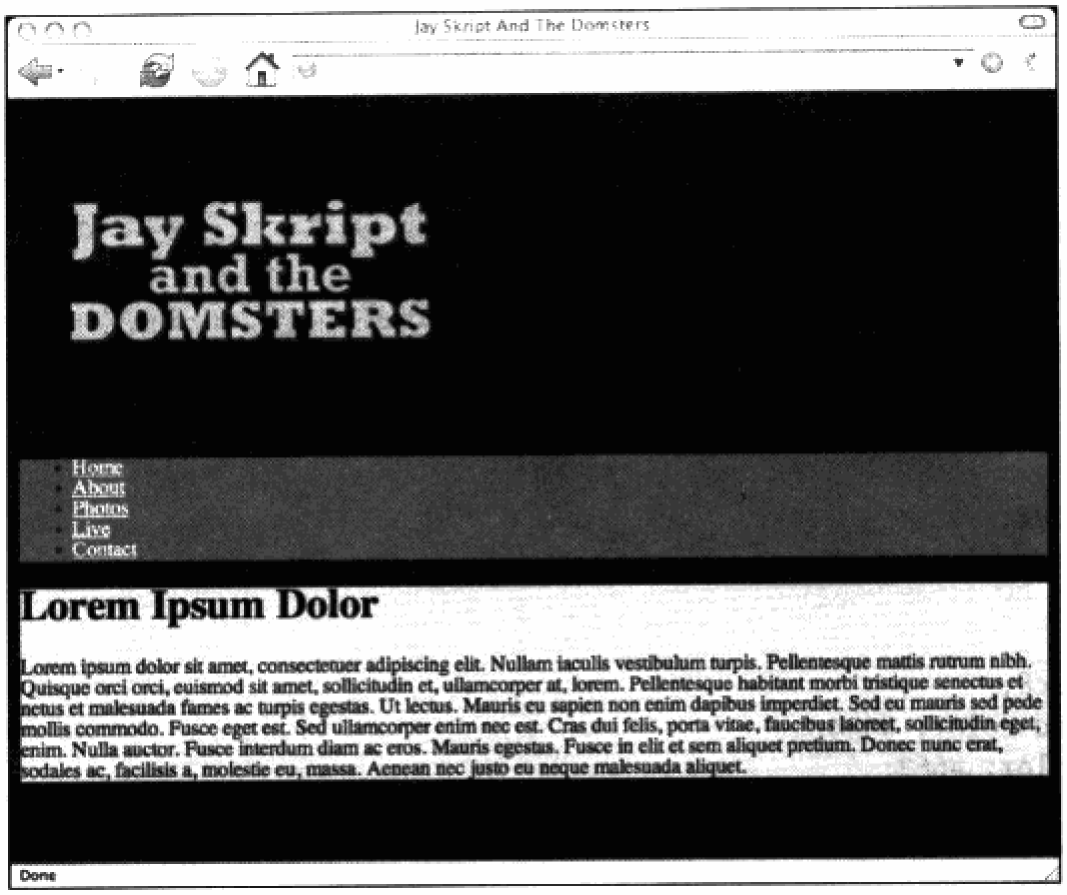
11.4 颜色

color.css 样式表应该是最不复杂的了。我在这里要特别提醒大家一句：如果你为某个元素设置了一种前景颜色，就应该再为它设置一种背景颜色。如果不注意这个细节，就有可能导致某些文本内容变成“隐形”文字的问题。

```
body {
  color: #fb5;
  background-color: #334;
}
a:link {
  color: #445;
  background-color: #eb6;
}
a:visited {
  color: #345;
  background-color: #eb6;
}
a:hover {
  color: #667;
  background-color: #fb5;
}
a:active {
  color: #778;
  background-color: #ec8;
}
#header {
  color: #ec8;
  background-color: #334;
  border-color: #667;
}
#navigation {
  color: #455;
  background-color: #789;
  border-color: #667;
}
#content {
  color: #223;
  background-color: #edc;
  border-color: #667;
}
#navigation ul {
  border-color: #99a;
}
#navigation a:link,#navigation a:visited {
  color: #eef;
  background-color: transparent;
  border-color: #99a;
```

```
}
#navigation a:hover {
  color: #445;
  background-color: #eb6;
}
#navigation a:active {
  color: #667;
  background-color: #ec8;
}
```

现在，模板文档 template.html 变得色彩丰富了。



11.4.1 布局

具体到这个例子，各个页面的基本布局相对也不是很复杂。页面上的所有内容都包含在同一栏里，不需要分栏显示。

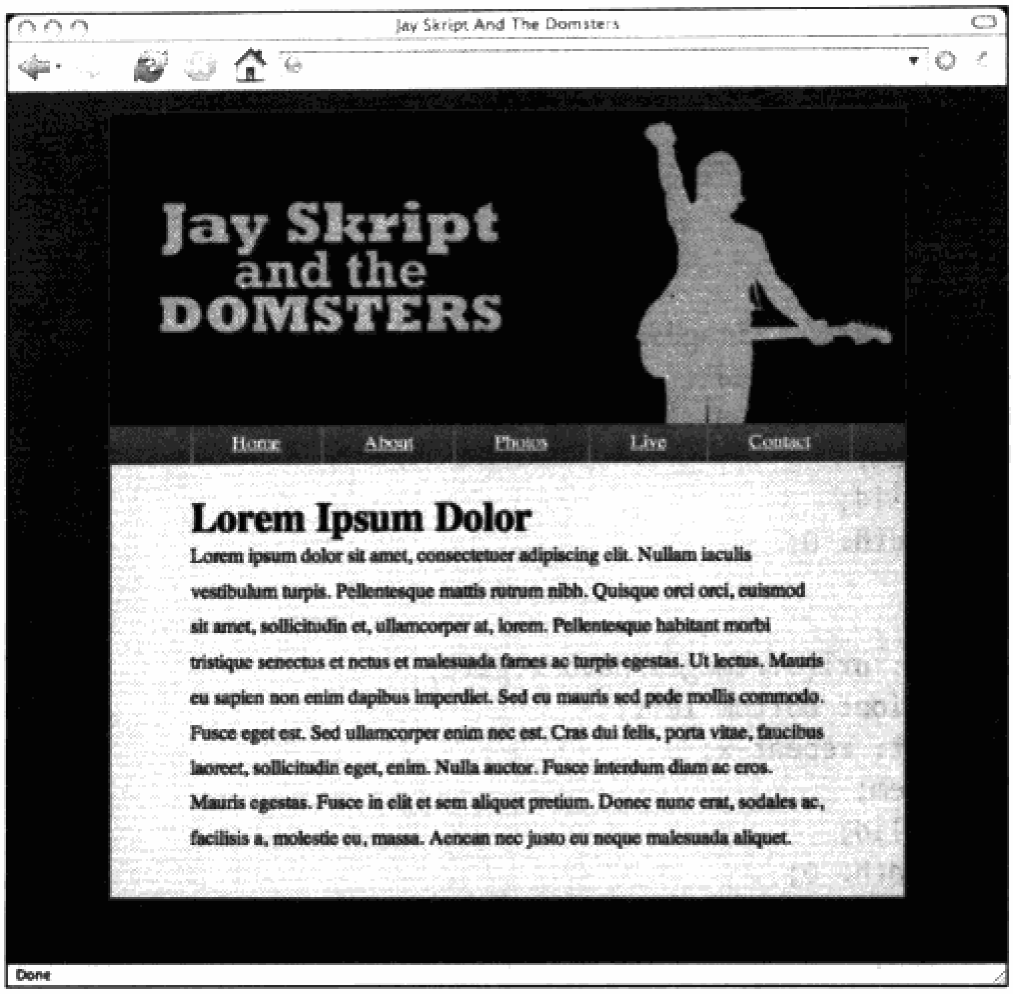
除了让导航条里的各个链接沿水平方向显示需要动点儿脑筋之外，layout.css 样式表里的东西都很简单。

请注意，如下所示的样式表一上来就用通配符把每个元素的匀空和间距设置成了零。这么做的好处是，可以让页面不受浏览器的页边距/字间距默认设置的影响——不同的浏览器往往有着不同的默认设置，而那会影响到页面的显示效果。把这些值设置为零等于是为自己开辟出了一个干净的“战场”：

```
* {
  padding: 0;
  margin: 0;
}
body {
  margin: 1em 10%;
  background-image: url(../images/background.gif);
  background-attachment: fixed;
  background-position: top left;
  background-repeat: repeat-x;
  max-width: 80em;
}
#header {
  background-image: url(../images/guitarist.gif);
  background-repeat: no-repeat;
  background-position: bottom right;
  border-width: .1em;
  border-style: solid;
  border-bottom-width: 0;
}
#navigation {
  background-image: url(../images/navbar.gif);
  background-position: bottom left;
  background-repeat: repeat-x;
  border-width: .1em;
  border-style: solid;
  border-bottom-width: 0;
  border-top-width: 0;
  padding-left: 10%;
}
#navigation ul {
  width: 100%;
  overflow: hidden;
  border-left-width: .1em;
  border-left-style: solid;
}
#navigation li {
  display: inline;
}
#navigation li a {
  display: block;
  float: left;
  padding: .5em 2em;
  border-right: .1em solid;
}
#content {
  border-width: .1em;
  border-style: solid;
  border-top-width: 0;
```

```
    padding: 2em 10%;
    line-height: 1.8em;
  }
```

下面是在我们通过 CSS 对它的颜色和布局做出设定之后模板文档 template.html 的样子。



11.4.2　字型

有时，把样式声明语句放到哪个 CSS 文件里并不是个容易做出的决定。比如说，对字体和字号的定义显然应该在 typography.css 文件里做出，可是对匀空和间距的声明应该放到哪个 CSS 文件里才是最合适的呢？说它们与页面布局有关也好，说它们与字体字号有关也罢；两种意见都有道理。具体到这个例子，我已经替你做出了决定：把匀空信息放在 layout.css 文件里，把间距信息放在 typography.css 文件里：

```
body {
  font-size: 76%;
  font-family: "Helvetica","Arial",sans-serif;
}
body * {
  font-size: 1em;
}
a {
  font-weight: bold;
```

```
  text-decoration: none;
}
#navigation {
  font-family: "Lucida Grande","Helvetica","Arial",sans-serif;
}
#navigation a {
  text-decoration: none;
  font-weight: bold;
}
#content {
  line-height: 1.8em;
}
#content p {
  margin: 1em 0;
}
h1 {
  font-family: "Georgia","Times New Roman",sans-serif;
  font: 2.4em normal;
}
h2 {
  font-family: "Georgia","Times New Roman",sans-serif;
  font: 1.8em normal;
  margin-top: 1em;
}
h3 {
  font-family: "Georgia","Times New Roman",sans-serif;
  font: 1.4em normal;
  margin-top: 1em;
}
```

下面是在我们把颜色、布局和字型等样式全部设定之后模板文档 template.html 的样子。



把 color.css、layout.css 和 typography.css 等三个文件放到 basic.css 样式表所在的 styles 子目录里去。

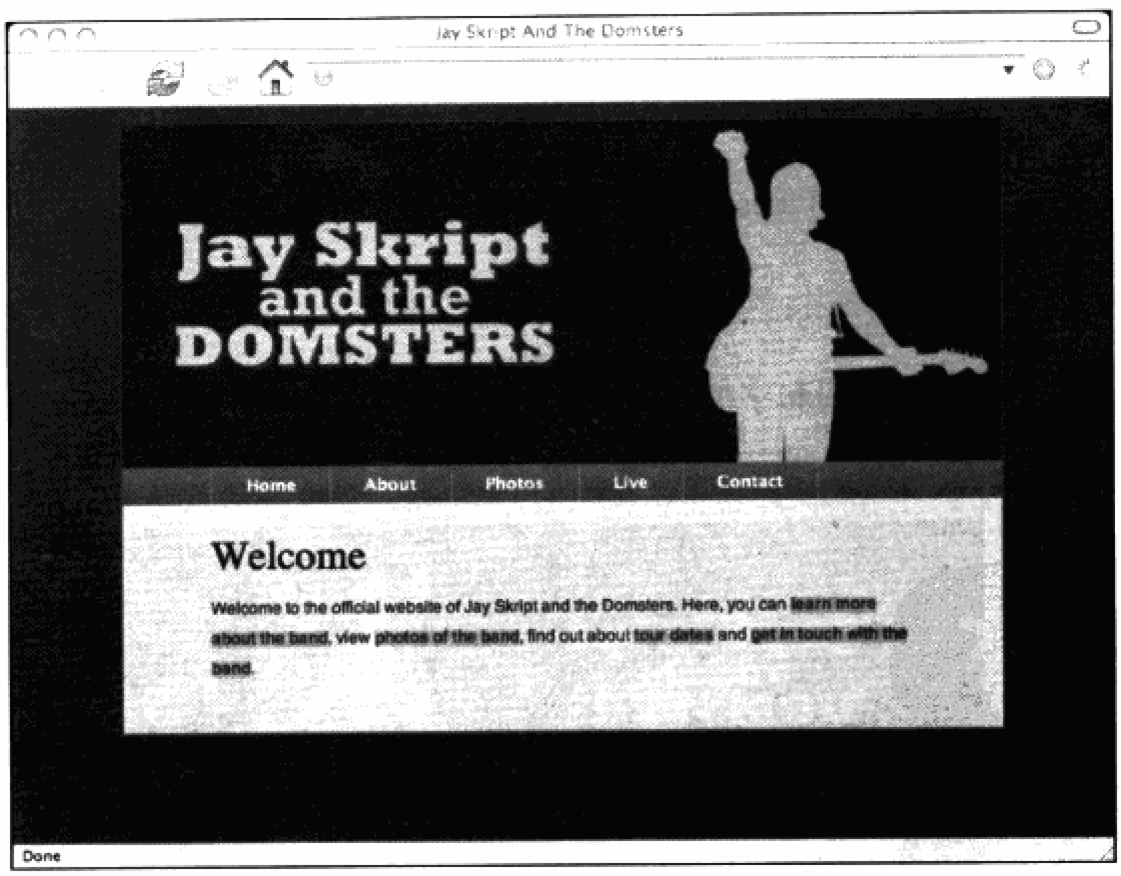
11.5　XHTML 文档

模板现在已经很符合你的预期了。样式表都工作得很好。你可以开始编写这个网站的页面了。

就从对应于网站主页的 index.html 文件开始好了。在这个页面上，id 属性值等于 content 的 div 元素里包含着一段介绍性文字：

```
<p id="intro">
Welcome to the official website of Jay Skript and the Domsters.
Here, you can <a href="about.html">learn more about the band</a>,
view <a href="photos.html">photos of the band</a>,
find out about <a href="live.html">tour dates</a>
and <a href="contact.html">get in touch with the band</a>.
</p>
```

下面是这个网站的主页在浏览器里的显示效果。



这个文本段的id属性值是intro。你可以利用它把某种特殊的样式施加在这段文本上。你还可以把它用做DOM脚本里的“挂钩”。

11.6　JavaScript 脚本

开始编写 DOM 代码之前，应该先把如何管理 JavaScript 脚本的问题安排妥当。

在即将编写的 JavaScript 函数当中，有些只适用于某一个页面，另一些则会被反复多次地用到。请把那些会被反复用到的函数集中全部放入 global.js 文件。

你肯定会多次用到 addLoadEvent()函数。每当你编写出一个将在文档加载完毕时得到执行的函数，你都需要用 addLoadEvent()函数把它绑定到 window.onload 事件处理函数上去：

```
function addLoadEvent(func) {
  var oldonload = window.onload;
  if (typeof window.onload != 'function') {
    window.onload = func;
  } else {
    window.onload = function() {
      oldonload();
      func();
    }
  }
}
```

insertAfter()也是一个常用的辅助函数。它可以弥补 insertBefore()函数的不足：

```
function insertAfter(newElement,targetElement) {
  var parent = targetElement.parentNode;
  if (parent.lastChild == targetElement) {
    parent.appendChild(newElement);
  } else {
    parent.insertBefore(newElement,targetElement.nextSibling);
  }
}
```

你还应该把在本书第 9 章里编写的 addClass()函数添加到 global.js 文件里去：

```
function addClass(element,value) {
  if (!element.className) {
    element.className = value;
  } else {
    newClassName = element.className;
    newClassName+= " ";
    newClassName+= value;
    element.className = newClassName;
  }
}
```

把 global.js 文件存放到 scripts 子目录里。在每个页面的<head>部分插入一个<script>标签来调用这个文件：

```
<script type="text/javascript" src="scripts/global.js"></script>
```

还有一个函数需要添加到 global.js 文件里去。

11.6.1 当前页面的标识

在从现有模板创建其他页面时，你将需要把一些 XHTML 内容插入到 id 属性值等于 content 的 div 元素里去。不同页面的文档中的这个部分互不相同。

从追求完美的角度讲，你还应该对 id 属性值等于 navigation 的 div 元素进行刷新——如果当前页面是（比如说）index.html，就没有必要在导航条里保留一个指向 index.html 文件的链接。

在实际工作中，不见得总是能对每个页面的导航条进行编辑。很多时候，包含着导航条的 XHTML 代码段会由一个服务器端脚本或命令（比如 include 命令）临时插入到页面文档里来。

我们不妨假设你正在创建的这个网站恰好属于这种情况。Web 服务器将使用一个服务器端脚本或命令把下面这段 XHTML 代码临时插入这个网站的各个页面：

```
<body>
  <div id="header">
    <img src="images/logo.gif" alt="Jay Skript and the Domsters" />
  </div>
  <div id="navigation">
    <ul>
      <li><a href="index.html">Home</a></li>
      <li><a href="about.html">About</a></li>
      <li><a href="photos.html">Photos</a></li>
      <li><a href="live.html">Live</a></li>
      <li><a href="contact.html">Contact</a></li>
    </ul>
  </div>
```

Apache Server Side Includes、PHP、ASP 和其他一些服务器端语言都支持这种做法。

这么做的好处是，可以把那些常用的(X)HTML 代码，比如页面的标题部分和导航条部分，集中存放到某个地方。当需要对页面标题或导航条进行刷新时，你只需在一个文件里完成修改，而用不着修改所有相关的页面文件。但这种做法的不利之处是，你将很难对页面的这个部分进行定制。

不管怎么说，当前页面应该带有某种明显的视觉标志。这可以让访问者一目了然地知道“我正在这里”。

修改 color.css 文件，为那些 class="here"的元素增加一组如下所示的样式声明：

```
#navigation a.here:link,
#navigation a.here:visited,
#navigation a.here:hover,
#navigation a.here:active {
  color: #eef;
  background-color: #799;
}
```

为了使用上述颜色，你将需要在导航条部分给指向当前页面的链接添加一个取值为 here 的 class 属性：

```
<a href="index.html" class="here">Home</a></li>
```

如果你使用了一个服务器端脚本或命令，为元素添加 class 属性可不那么容易。在理想的情况下，服务器端技术应该健壮到能够正确地把必要的(X)HTML 代码插入到各有关页面里的程度。但在实际工作中，事情却并非总是如此。

这正是 JavaScript 可以帮上大忙的地方。

此时，JavaScript 或许是最后一项补救措施了。只要把 class="here"直接插入到有关页面的(X)HTML 代码里，后面的事情就好办了。但千万要记住：只有当(X)HTML 代码不在你控制范围内时才应该考虑基于 JavaScript 的解决方案。

你应该编写一个 highlightPage()的函数来完成以下操作：

(1) 把导航条里的所有链接全部检索出来。

(2) 遍历这些链接。

(3) 如果能找到一个与当前 URL 相匹配的链接，就给它加上一个取值为 here 的 class 属性。

和往常一样，在这个函数的开头部分应该安排一些测试，以检查浏览器是否理解你将使用的 DOM 方法和属性。还应该测试一下那个 id 属性值等于 navigation 的 div 元素是否真的存在：

```
function highlightPage() {
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("navigation")) return false;
  var nav = document.getElementById("navigation");
```

接下来，提取所有的导航链接并用一个 for 循环去遍历它们：

```
var links = nav.getElementsByTagName("a");
for (var i=0; i<links.length; i++) {
```

在 for 循环里，把 links[i]元素的 URL 与当前页面的 URL 进行比较。links[i]元素的 URL 可以用 getAttribute("href")获得，当前页面的 URL 可以用 window.location.href 获得：

```
var linkurl = links[i].getAttribute("href");
var currenturl = window.location.href;
```

JavaScript 为我们准备了很多用来进行字符串比较的方法。indexOf()方法可以在一个字符串里寻找一个子串的位置：

```
string.indexOf(substring)
```

这个方法的返回值是子串 *substring* 在字符串 *string* 里第一次出现的位置。具体到这个例子，你只是想知道某个字符串是不是在另一个字符串里："links[i]元素的 URL 是不是在当前页面的 URL 里"：

```
currenturl.indexOf(linkurl)
```

如果没有找到一个匹配，indexOf()方法将返回-1；如果它返回的是其他值，就说明它找到了一个匹配。于是，只要 indexOf()方法返回的不是-1，就可以前进到这个函数的最后一步：

```
if (currenturl.indexOf(linkurl) != -1) {
```

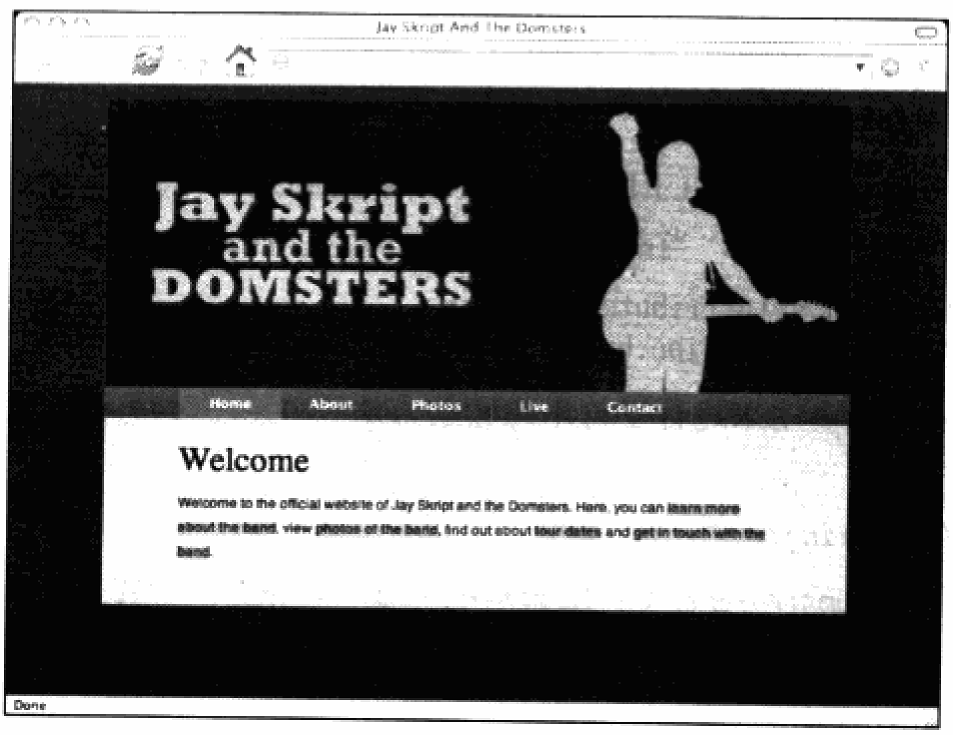
至此，links[i]元素所代表的链接肯定是指向当前页面的链接。给这个链接添加一个取值为 here 的 class 属性：

```
links[i].className = "here";
```

最后，先依次结束if语句、for循环和整个函数——即连续写出三个右花括号，然后用addLoadEvent()函数调用highlightPage()函数：

```
function highlightPage() {
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("navigation")) return false;
  var nav = document.getElementById("navigation");
  var links = nav.getElementsByTagName("a");
  for (var i=0; i<links.length; i++) {
    var linkurl = links[i].getAttribute("href");
    var currenturl = window.location.href;
    if (currenturl.indexOf(linkurl) != -1) {
      links[i].className = "here";
    }
  }
}
addLoadEvent(highlightPage);
```

把这个函数添加到 global.js 文件里。现在，如果重新加载 index.html 文件，就可以看到导航条里的"Home"链接呈现出"我是当前页面"的标识效果。



你还可以对 highlightPage()函数做些改进以达到一箭双雕的目的。

如果你能给每个页面的 body 元素分别设置一个唯一的 id 属性值，就可以利用这些 id 值为

不同的页面设置一些不同的专用样式。把当前链接(这个链接现在有了一个取值为"here"的 class 属性)的标识文本设置为当前页面的 id 属性值即可。下面这条语句用 JavaScript 提供的 toLowerCase()方法把提取出来的字符串转换成了小写字母：

```
var linktext = links[i].lastChild.nodeValue.toLowerCase();
```

上面这条语句将提取出当前链接的最后一个子节点的值——也就是当前链接的标识文本，并把它转换为小写字母。如果当前链接的标识文本是(比如说)"Home", linktext 变量的值就将是 home。接下来，把这个变量赋值给 body 元素的 id 属性即可：

```
document.body.setAttribute("id",linktext);
```

这相当于在<body>标签里写出 "id="home""。

下面是 highlightPage()函数现在的代码清单：

```
function highlightPage() {
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("navigation")) return false;
  var nav = document.getElementById("navigation");
  var links = nav.getElementsByTagName("a");
  for (var i=0; i<links.length; i++) {
    var linkurl = links[i].getAttribute("href");
    var currenturl = document.location.href;
    if (currenturl.indexOf(linkurl) != -1) {
      links[i].className = "here";
      var linktext = links[i].lastChild.nodeValue.toLowerCase();
      document.body.setAttribute("id",linktext);
    }
  }
}
addLoadEvent(highlightPage);
```

现在，index.html 文件中的 body 元素有了一个取值为 home 的 id 属性，about.html 文件中的 body 元素有了一个取值为 about 的 id 属性，photos.html 文件中的 body 元素有了一个取值为 photos 的 id 属性；依此类推。

你可以在 CSS 样式表里把这些新插入的标识符当做"挂钩"来使用。比如说，这个网站的每个页面都包含着一个 id="header"的 div 元素，而你可以利用新插入的标识符为这个 div 元素另外指定一张背景图片。

先为每个网页创建一张 250 像素×250 像素的图片。这些图片我已经为你准备好了：lineup.gif、basshead.gif、bassist.gif 和 drummer.gif。请把这些文件放到 images 文件夹里。

现在，你可以在 layout.css 文件里用下面这些 CSS 语句给网页标题部分加上背景图片装饰了：

```
#about #header {
  background-image: url(../images/lineup.gif);
```

```
}
#photos #header {
  background-image: url(../images/basshead.gif);
}
#live #header {
  background-image: url(../images/bassist.gif);
}
#contact #header {
  background-image: url(../images/drummer.gif);
}
```

现在，每个网页的标题部分都有了自己的图片背景。

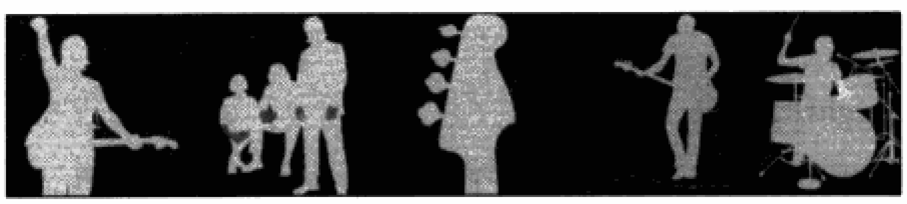
11.6.2 JavaScript 动画

主页就要有特色。主页是绝大多数访问者在网站上看到的第一个页面，如果你想让人喜欢你的网站，就应该在主页上多下点儿工夫。把我们在本书第 8 章编写的 JavaScript 动画脚本用在这里是再合适不过的了。

id 属性是 intro 的文本段包含着指向网站其他页面的所有链接。当访问者把鼠标指针悬停在其中某个链接的上方时，用一种视觉印象让他们了解等待在那里的内容会是个很不错的主意。刚才已经为每个页面的标题部分准备了一张背景图片，现在可以把那些图片的缩微版本显示给访问者。

先把那些背景图片统一缩放到 150 像素×150 像素的大小，然后将其拼接成一张 750 像素×150 像素的图片并把它保存为 slideshow.gif 文件。把这张图片放到 images 文件夹里。

拼接出来的图片应该是下面这个样子。



在 scripts 文件夹里创建一个名为 home.js 的 JavaScript 脚本文件，用来生成动画效果的 JavaScript 函数将保存在这里。这个动画将只出现在网站的主页上，所以只需要在 index.html 文档里调用它。

为了获得动画效果，你将需要我们在本书第 10 章编写的 moveElement()函数：

```
function moveElement(elementID,final_x,final_y,interval) {
  if (!document.getElementById) return false;
  if (!document.getElementById(elementID)) return false;
  var elem = document.getElementById(elementID);
  if (elem.movement) {
    clearTimeout(elem.movement);
  }
  if (!elem.style.left) {
```

```
      elem.style.left = "0px";
    }
    if (!elem.style.top) {
      elem.style.top = "0px";
    }
    var xpos = parseInt(elem.style.left);
    var ypos = parseInt(elem.style.top);
    if (xpos == final_x && ypos == final_y) {
      return true;
    }
    if (xpos < final_x) {
      var dist = Math.ceil((final_x - xpos)/10);
      xpos = xpos + dist;
    }
    if (xpos > final_x) {
      var dist = Math.ceil((xpos - final_x)/10);
      xpos = xpos - dist;
    }
    if (ypos < final_y) {
      var dist = Math.ceil((final_y - ypos)/10);
      ypos = ypos + dist;
    }
    if (ypos > final_y) {
      var dist = Math.ceil((ypos - final_y)/10);
      ypos = ypos - dist;
    }
    elem.style.left = xpos + "px";
    elem.style.top = ypos + "px";
    var repeat =
➡ "moveElement('"+elementID+"',"+final_x+","+final_y+","+interval+")";
    elem.movement = setTimeout(repeat,interval);
  }
```

现在，你需要在主页上创建一些动画元素，并对有关的链接做出必要的初始设置。此时，我们不妨假设你决定把动画安排在“intro”文本段的紧后面：

```
  function prepareSlideshow() {
    if (!document.getElementsByTagName) return false;
    if (!document.getElementById) return false;
    if (!document.getElementById("intro")) return false;
    var intro = document.getElementById("intro");
    var slideshow = document.createElement("div");
    slideshow.setAttribute("id","slideshow");
  var preview = document.createElement("img");
  preview.setAttribute("src","images/slideshow.gif");
  preview.setAttribute("alt","a glimpse of what awaits you");
  preview.setAttribute("id","preview");
  slideshow.appendChild(preview);
  insertAfter(slideshow,intro);
```

接下来，遍历“intro”文本段里的所有链接。你需要根据鼠标指针正悬停在哪个链接的上方来移动 preview 元素。比如说，如果当前链接（this）的 href 属性值是字符串“about.html”，就要把 preview 元素移动–150 像素；如果 href 属性值是字符串“photos.html”，就要把 preview 元素移动–300 像素；依此类推。

为了让动画效果更显眼，可以把传递给 moveElement()函数的 interval 参数值设置为 5 毫秒：

```
  var links = intro.getElementsByTagName("a");
  for (var i=0; i<links.length; i++) {
    links[i].onmouseover = function() {
      var destination = this.getAttribute("href");
      if (destination.indexOf("index.html") != -1) {
        moveElement("preview",0,0,5);
      }
      if (destination.indexOf("about.html") != -1) {
        moveElement("preview",-150,0,5);
      }
      if (destination.indexOf("photos.html") != -1) {
        moveElement("preview",-300,0,5);
      }
      if (destination.indexOf("live.html") != -1) {
        moveElement("preview",-450,0,5);
      }
      if (destination.indexOf("contact.html") != -1) {
        moveElement("preview",-600,0,5);
      }
    }
  }
}
```

用 addLoadEvent()函数调用 prepareSlideShow()函数：

```
addLoadEvent(prepareSlideshow);
```

把以上代码保存为 home.js 文件。你将需要在 index.html 文档的<head>部分增加一个<script>标签来调用这个 JavaScript 脚本文件：

```
<script type="text/javascript" src="scripts/home.js"></script>
```

你还需要对样式进行刷新——把以下代码添加到 layout.css 文件里：

```
#slideshow {
  width: 150px;
  height: 150px;
  position: relative;
  overflow: hidden;
}
```

在一个 Web 浏览器里刷新 index.html 文档，就可以看到动画效果了。

现在的动画效果看起来已经相当不错，但我们还可以让它更好一点儿：给动画加上一个“窗框”。

创建一个150像素×150像素大小的图片，让它几乎透明，并在它的四个角上用 id="content" 的那个 div 元素的背景颜色制造出圆角效果。把这张图片保存为 frame.gif 文件并把它存放到 images 文件夹里。

首先，在 home.js 文件中的 prepareSlideShow()函数里，把以下代码插入到 slideshow 元素创建语句的紧后面：

```
var frame = document.createElement("img");
frame.setAttribute("src","images/frame.gif");
frame.setAttribute("alt","");
frame.setAttribute("id","frame");
slideshow.appendChild(frame);
```

然后，为了确保这个容器“遮盖”在动画图片的上方，在 layout.css 文件里增加以下语句：

```
#frame {
  position: absolute;
  top: 0;
  left: 0;
  z-index: 99;
}
```

在 Web 浏览器里刷新 index.html 文件，就可以看到带有窗框的动画效果了。

现在，当访问者把鼠标指针悬停在“intro”文本段中的某个链接的上方时，他就可以看到动画效果了。如果你愿意，还可以让导航条（对应于 id="navigation"的那个 div 元素）里的链接也具备这样的动画功能。这只需稍稍修改一下下面这条语句：

```
var links = intro.getElementsByTagName("a");
```

修改为：

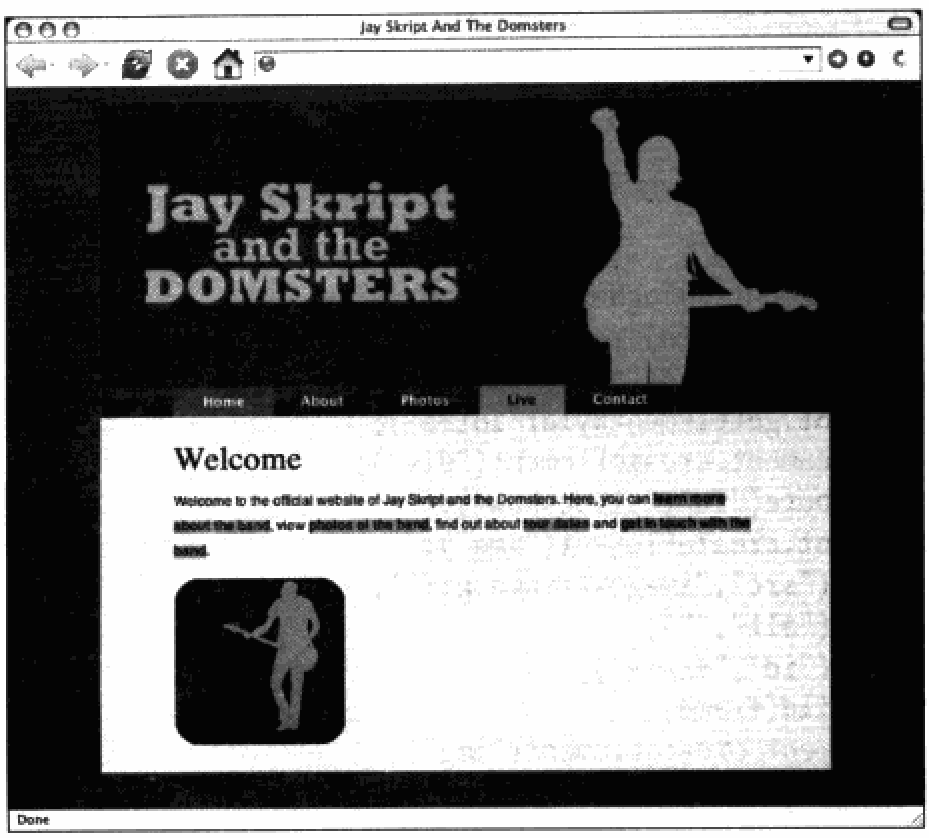
```
var links = document.getElementsByTagName("a");
```

下面是最终完成的 prepareSlideShow()函数的代码清单：

```
function prepareSlideshow() {
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("intro")) return false;
  var intro = document.getElementById("intro");
  var slideshow = document.createElement("div");
  slideshow.setAttribute("id","slideshow");
  var frame = document.createElement("img");
  frame.setAttribute("src","images/frame.gif");
  frame.setAttribute("alt","");
  frame.setAttribute("id","frame");
  slideshow.appendChild(frame);
  var preview = document.createElement("img");
  preview.setAttribute("src","images/slideshow.gif");
```

```
    preview.setAttribute("alt","a glimpse of what awaits you");
    preview.setAttribute("id","preview");
    slideshow.appendChild(preview);
    insertAfter(slideshow,intro);
    var links = document.getElementsByTagName("a");
    for (var i=0; i<links.length; i++) {
      links[i].onmouseover = function() {
        var destination = this.getAttribute("href");
        if (destination.indexOf("index.html") != -1) {
          moveElement("preview",0,0,5);
        }
        if (destination.indexOf("about.html") != -1) {
          moveElement("preview",-150,0,5);
        }
        if (destination.indexOf("photos.html") != -1) {
          moveElement("preview",-300,0,5);
        }
        if (destination.indexOf("live.html") != -1) {
          moveElement("preview",-450,0,5);
        }
        if (destination.indexOf("contact.html") != -1) {
          moveElement("preview",-600,0,5);
        }
      }
    }
  }
  addLoadEvent(prepareSlideshow);
```

现在，当你把鼠标指针悬停在导航条里的任何一个链接的上方时，动画效果就会被触发。

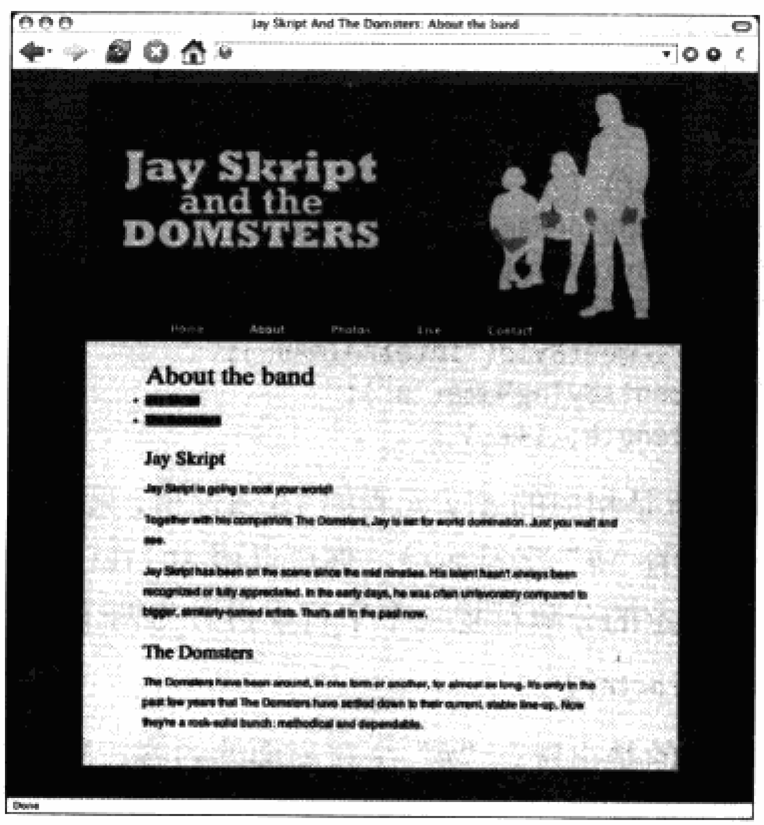


11.6.3 内部浏览

这个网站的下一个页面是“About”页面。把以下 XHTML 代码插入到 template.html 文件中 id 属性是 content 的那个 div 元素里：

```
<h1>About the band</h1>
<ul id="internalnav">
  <li><a href="#jay">Jay Skript</a></li>
  <li><a href="#domsters">The Domsters</a></li>
</ul>
<div class="section" id="jay">
  <h2>Jay Skript</h2>
  <p>Jay Skript is going to rock your world!</p>
  <p>Together with his compatriots The Domsters,
Jay is set for world domination. Just you wait and see.</p>
  <p>Jay Skript has been on the scene since the mid nineties.
His talent hasn't always been recognized or fully appreciated.
In the early days, he was often unfavorably compared to bigger,
similarly-named artists. That's all in the past now.</p>
</div>
<div class="section" id="domsters">
  <h2>The Domsters</h2>
  <p>The Domsters have been around, in one form or another,
for almost as long. It's only in the past few years that The Domsters
have settled down to their current, stable line-up.
Now they're a rock-solid bunch: methodical and dependable.</p>
</div>
```

把整个文件另存为 about.html 文件。下面是“About”页面在浏览器里现在的显示效果。



这个页面本身没有任何问题。但美中不足的是它好像太长了点儿，所以我在这个页面的开头部分增加了一个 id="internalnav"的 ul 元素，并在其中添加了几个内部链接。这些链接中的每个分别指向某个 class="section"的 div 元素。

利用 JavaScript 和 DOM，你可以有选择地显示和隐藏那几个 class="section"的 div 元素，让它们在某一时刻只有一个在浏览器窗口里是可见的。

下面这个函数将只把那几个 class="section"的 div 元素当中有着特定 id 属性值的那个显示在浏览器窗口里，其他的同类 div 元素在浏览器窗口都不可见：

```
function showSection(id) {
  var divs = document.getElementsByTagName("div");
  for (var i=0; i<divs.length; i++ ) {
    if (divs[i].className.indexOf("section") == -1) continue;
    if (divs[i].getAttribute("id") != id) {
      divs[i].style.display = "none";
    } else {
      divs[i].style.display = "block";
    }
  }
}
```

如上所示，showSection()函数将刷新那几个 class="section"的 div 元素的 display 样式属性：除了那个有着特定 id 属性值的 div 元素外，它将把其他几个 div 元素的 dispaly 属性都设置为 none。在调用这个函数时，你需要向它传递一个 id 属性值作为参数，它将把有着这个 id 属性值的 div 元素的 display 属性设置为“block”。

你需要让 showSection()函数在 internalnav 清单里的任何一个链接被点击时都将得到执行。

我们来编写一个 prepareInterNav()的函数。这个函数将对“internalnav”清单里的所有链接进行遍历：

```
function prepareInternalnav() {
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("internalnav")) return false;
  var nav = document.getElementById("internalnav");
  var links = nav.getElementsByTagName("a");
  for (var i=0; i<links.length; i++ ) {
```

每个链接的 href 属性值其实就是相应的 div 元素的 id 属性值，只不过在它们的开头多了一个用来表明“这是一个内部链接”的“#”字符而已。你可以用 split()方法把那个 div 元素的 id 属性提取出来，它是根据某个给定的分隔符把一个字符串拆分为两个或更多个部分的常用方法：

```
array = string.split(character)
```

具体到这个例子，你需要的是出现在“#”字符后面的内容，所以你需要用“#”字符来拆分

href 属性的值。如此得到的数组将包含着两个字符串值：第一个是出现在“#”字符前面的内容，第二是出现在“#”字符后面的内容。记住，在 JavaScript 语言里，数组中的第一个元素的下标永远是零。你现在感兴趣的是数组中的第二个元素，它的下标是 1：

```
var sectionId = links[i].getAttribute("href").split("#")[1];
```

上面这条语句将把第一个“#”字符后面的内容提取到变量 sectionId 里去。

接下来，检查文档里是否实际存在着一个以 sectionId 作为其 id 标识的元素。如果不存在，则立刻转入下一次循环：

```
if (!document.getElementById(sectionId)) continue;
```

在页面加载时，class 属性值是 section 的所有 div 元素都应该是不可见的。这种默认效果可以用下面这条语句来获得：

```
document.getElementById(sectionId).style.display = "none";
```

接下来，你需要为各有关链接增加这样一个 onclick 事件处理函数：当其中的某个链接被点击时，把变量 sectionId 作为参数传递给 showSection()函数。

这里遇到了一个与变量的作用域有关的问题。变量 sectionId 是 prepareInterNav()函数中的一个局部变量，它只存在于这个函数的上下文里——它在 onclick 事件处理函数里根本不存在。

你可以通过给各有关链接增加一个定制属性的办法来解决这个问题：把变量 sectionId 的值赋给一个名为“destination”的定制属性：

```
links[i].destination = sectionId;
```

destination 属性有着你需要的作用域。现在可以从 onclick 事件处理函数去查询这个属性了：

```
links[i].onclick = function() {
  showSection(this.destination);
  return false;
}
```

用一些右花括号来结束 prepareInterNav()函数，然后用 addLoadEvent()函数调用它：

```
addLoadEvent(prepareInternalnav);
```

把 showSection()和 prepareInterNav()函数保存到 scripts 文件夹里的一个名为 about.js 的文件里去：

```
function showSection(id) {
  var divs = document.getElementsByTagName("div");
  for (var i=0; i<divs.length; i++ ) {
    if (divs[i].className.indexOf("section") == -1) continue;
    if (divs[i].getAttribute("id") != id) {
      divs[i].style.display = "none";
    } else {
```

```
      divs[i].style.display = "block";
    }
  }
}

function prepareInternalnav() {
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("internalnav")) return false;
  var nav = document.getElementById("internalnav");
  var links = nav.getElementsByTagName("a");
  for (var i=0; i<links.length; i++ ) {
    var sectionId = links[i].getAttribute("href").split("#")[1];
    if (!document.getElementById(sectionId)) continue;
    document.getElementById(sectionId).style.display = "none";
    links[i].destination = sectionId;
    links[i].onclick = function() {
      showSection(this.destination);
      return false;
    }
  }
}

addLoadEvent(prepareInternalnav);
```

在 about.html 文档里的<head>部分增加一个如下所示的<script>标签来调用这个文件：

```
<script type="text/javascript" src="scripts/about.js"></script>
```

现在，把 about.html 文档加载到一个 Web 浏览器里以测试这两个函数的工作情况。当你点击这个“about”页面里的某个内部链接时，你应该只看到与之对应的段落内容，如下所示。



页面越长，这个函数的好处就越明显。比如说，如果你正在编写一个“常见问题答疑”页面，则可以把每个问题设为一个内部链接，然后利用本节介绍的技巧让用户在点击某个问题时只能看到那个问题的答案，而其他问题的答案都“深藏不露”。

11.6.4 JavaScript 美术馆

接下来将要编写的是 photos.html 文件。把在前面的有关章节里编写和优化的“JavaScript 美术馆”案例套用在这里是再合适不过的了。

客户已经向你提供了 4 张尺寸都是 400 像素×300 像素的“Jay Skript and the Domsters”乐队的演出照片，它们现在就存放在以下几个文件里：

- concert.jpg
- bassist.jpg
- guitarist.jpg
- crowd.jpg

在 images 文件夹再创建一个 photos 文件夹，把这 4 张图片放到 photos 文件夹里。

先为每张图片制作一个 100 像素×100 像素的缩微图：

- thumbnail_ concert.jpg
- thumbnail_ bassist.jpg
- thumbnail_ guitarist.jpg
- thumbnail_ crowd.jpg

把这几个文件也放到 photos 文件夹里。

接下来，创建一个链接清单（ul 元素），让清单里的每个链接分别指向一张全尺寸的乐队演出照片；把这个清单元素的 id 属性设置为 imagegallery。在每个链接里插入一个<img>标签，把每张图片的 src 属性设置为该图片的缩微图：

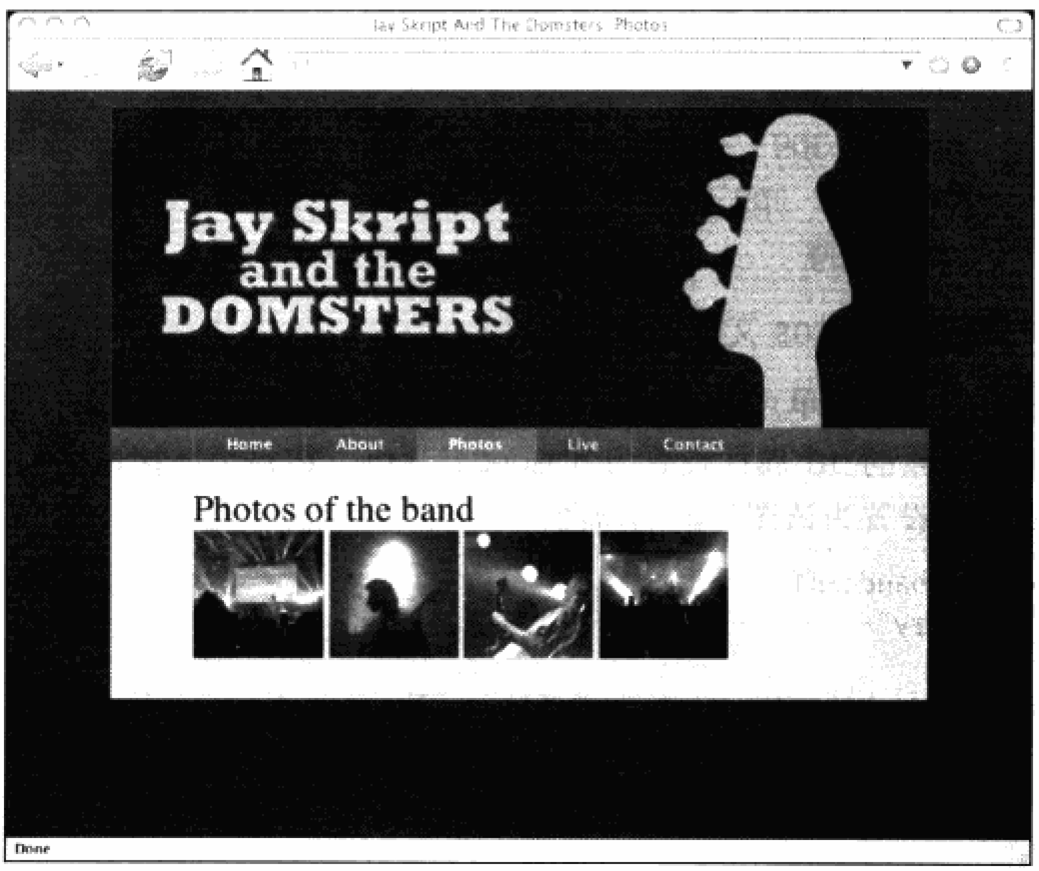
```
<h1>Photos of the band</h1>
<ul id="imagegallery">
  <li>
    <a href="images/photos/concert.jpg" title="The crowd goes wild">
      <img src="images/photos/thumbnail_concert.jpg"
➥ alt="the band in concert" />
    </a>
  </li>
  <li>
    <a href="images/photos/bassist.jpg" title="An atmospheric moment">
      <img src="images/photos/thumbnail_bassist.jpg"
➥ alt="the bassist" />
    </a>
  </li>
```

```
    <li>
      <a href="images/photos/guitarist.jpg" title="Rocking out">
        <img src="images/photos/thumbnail_guitarist.jpg"
➥ alt="the guitarist" />
      </a>
    </li>
    <li>
      <a href="images/photos/crowd.jpg" title="Encore! Encore!">
        <img src="images/photos/thumbnail_crowd.jpg"
➥ alt="the audience" />
      </a>
    </li>
  </ul>
```

编辑 template.html 文件，把上面这些代码插入 id 属性是 content 的那个 div 元素里，然后把整个文件另存为 photos.html。

刷新 layout、css 文件，让那些缩微图沿水平方向而不是垂直方向排列：

```
#imagegallery li {
  display: inline;
}
```



你还需要多制作一张图片作为“占位符”。把“占位符”图片命名为 placeholder.gif 并把这个文件存放到 images 文件夹里。

现在，把我们在本书第 6 和第 7 章里编写的“JavaScript 美术馆”脚本集中到 photos.js 文件，再把这个文件存放入 scripts 文件夹：

```
function showPic(whichpic) {
  if (!document.getElementById("placeholder")) return true;
  var source = whichpic.getAttribute("href");
  var placeholder = document.getElementById("placeholder");
  placeholder.setAttribute("src",source);
  if (!document.getElementById("description")) return false;
  if (whichpic.getAttribute("title")) {
    var text = whichpic.getAttribute("title");
  } else {
    var text = "";
  }
  var description = document.getElementById("description");
  if (description.firstChild.nodeType == 3) {
    description.firstChild.nodeValue = text;
  }
  return false;
}

function preparePlaceholder() {
  if (!document.createElement) return false;
  if (!document.createTextNode) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("imagegallery")) return false;
  var placeholder = document.createElement("img");
  placeholder.setAttribute("id","placeholder");
  placeholder.setAttribute("src","images/placeholder.gif");
  placeholder.setAttribute("alt","my image gallery");
  var description = document.createElement("p");
  description.setAttribute("id","description");
  var desctext = document.createTextNode("Choose an image");
  description.appendChild(desctext);
  var gallery = document.getElementById("imagegallery");
  insertAfter(description,gallery);
  insertAfter(placeholder,description);
}

function prepareGallery() {
  if (!document.getElementsByTagName) return false;
  if (!document.getElementById) return false;
  if (!document.getElementById("imagegallery")) return false;
  var gallery = document.getElementById("imagegallery");
  var links = gallery.getElementsByTagName("a");
  for ( var i=0; i < links.length; i++) {
    links[i].onclick = function() {
      return showPic(this);
    }
  }
}
```

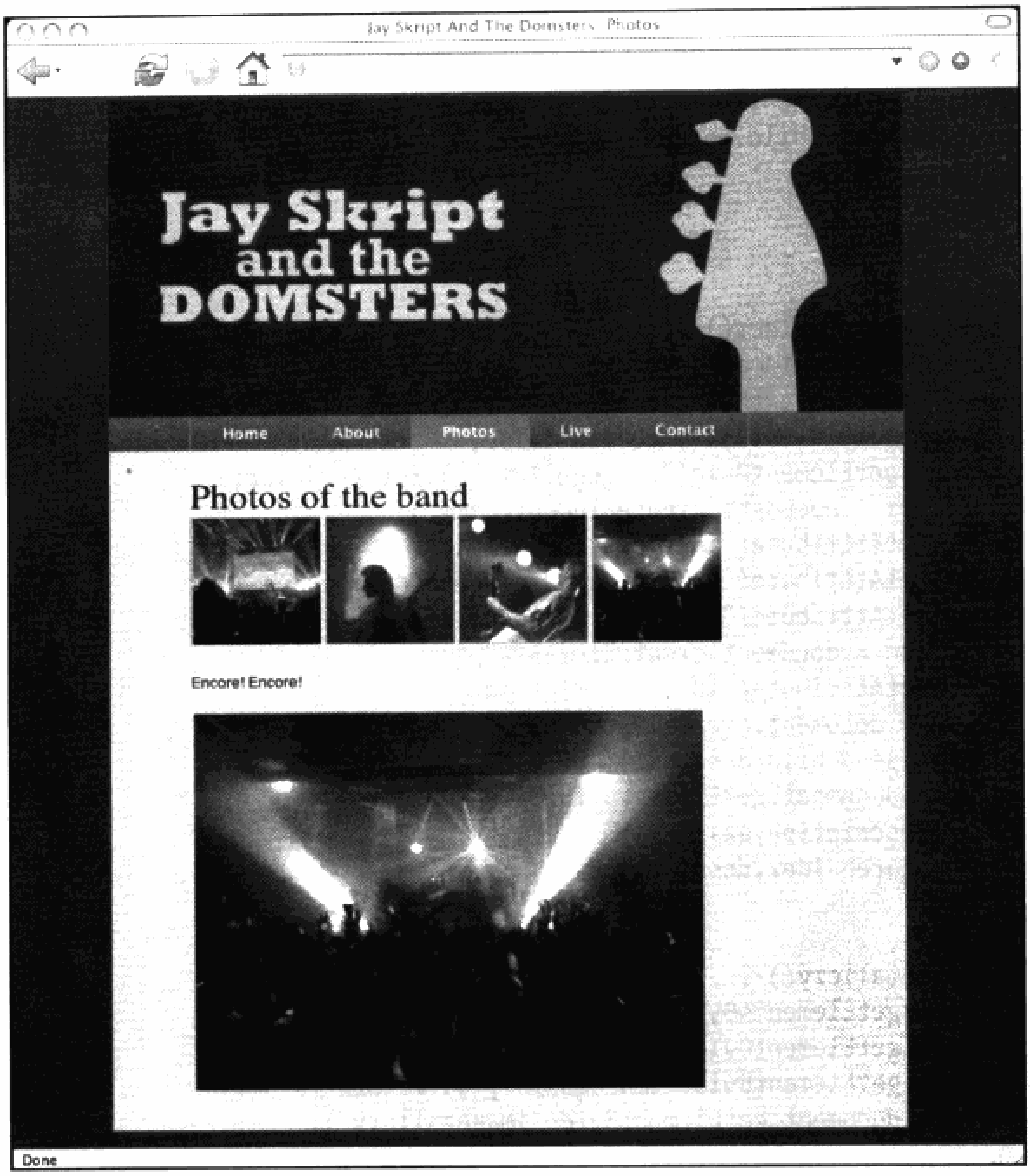
```
addLoadEvent(preparePlaceholder);
addLoadEvent(prepareGallery);
```

这里只有一处小改动：description 文本现在被挪到了 placeholder 图片的前面。

在 photos.html 文件里插入一个如下所示的<script>标签来调用 photos.js 文件：

```
<script type="text/javascript" src="scripts/photos.js"></script>
```

把 photos.html 文件加载到 Web 浏览器里，就可以看到这个图片页面的效果了。



11.6.5　改进表格

你的客户还向你提供了一份“Jay Skript and the Domsters”乐队的演出日程表。这份日程表列出了每场演唱会的日期、所在城市和演出地点。这是典型的表格型数据，所以你决定在“Live”页面里用一个<table>标签来插入这份演出日程表：

```
<h1>Tour dates</h1>
<table summary="when and where you can see the band">
  <thead>
  <tr>
    <th>Date</th>
    <th>City</th>
    <th>Venue</th>
  </tr>
  </thead>
  <tbody>
  <tr>
    <td>June 9th</td>
    <td>Portland, <abbr title="Oregon">OR</abbr></td>
    <td>Crystal Ballroom</td>
  </tr>
  <tr>
    <td>June 10th</td>
    <td>Seattle, <abbr title="Washington">WA</abbr></td>
    <td>Crocodile Cafe</td>
  </tr>
  <tr>
    <td>June 12th</td>
    <td>Sacramento, <abbr title="California">CA</abbr></td>
    <td>Torch Club</td>
  </tr>
  <tr>
    <td>June 17th</td>
    <td>Austin, <abbr title="Texas">TX</abbr></td>
    <td>Speakeasy</td>
  </tr>
  </tbody>
</table>
```

编辑 template.html 文件，把上面这些代码插入 id 属性是 content 的那个 div 元素里，然后把整个文件另存为 live.html。

刷新 layout.css 文件，增加一些针对表格项的样式：

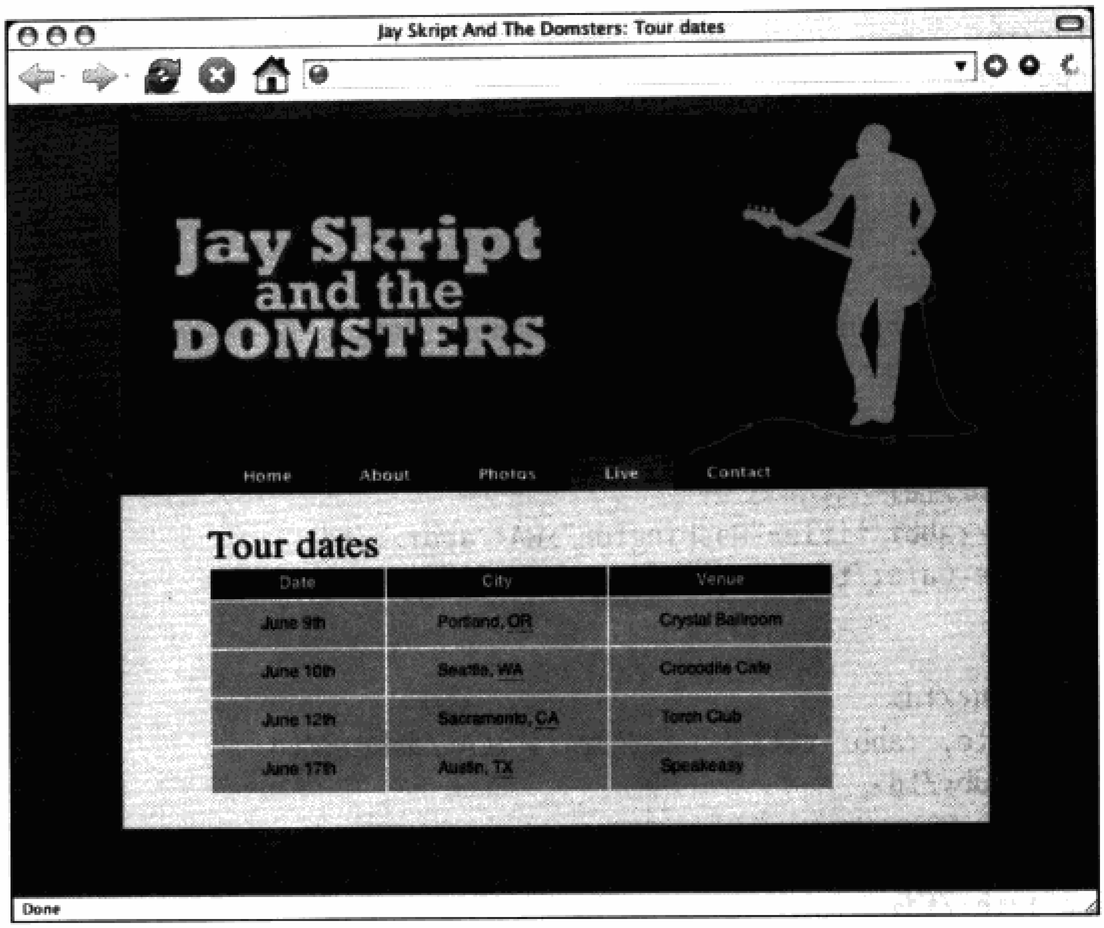
```
td {
  padding: .5em 3em;
}
```

刷新 color.css 文件，增加一些针对表格标题和表格行的颜色样式：

```
th {
  color: #edc;
  background-color: #455;
}
tr td {
  color: #223;
```

```
    background-color: #eb6;
  }
```

现在，如果把 live.html 文件加载到一个 Web 浏览器里，就可以看到一个在网上很常见的表格了。



这是利用我们在本书第 9 章编写的 stripeTables()和 highlightRows()函数设置表格样式的绝好机会。你还可以把 displayAbbreviations()函数用在这里。

把这几个函数全部写入 live.js 文件，在文件的末尾加上必要的 addLoadEvent()函数调用语句：

```
function stripeTables() {
  if (!document.getElementsByTagName) return false;
  var tables = document.getElementsByTagName("table");
  for (var i=0; i<tables.length; i++) {
    var odd = false;
    var rows = tables[i].getElementsByTagName("tr");
    for (var j=0; j<rows.length; j++) {
      if (odd == true) {
        addClass(rows[j],"odd");
        odd = false;
      } else {
        odd = true;
      }
    }
  }
}
```

```
function highlightRows() {
  if(!document.getElementsByTagName) return false;
  var rows = document.getElementsByTagName("tr");
  for (var i=0; i<rows.length; i++) {
    rows[i].oldClassName = rows[i].className
    rows[i].onmouseover = function() {
      addClass(this,"highlight");
    }
    rows[i].onmouseout = function() {
      this.className = this.oldClassName
    }
  }
}

function displayAbbreviations() {
  if (!document.getElementsByTagName || !document.createElement
➥ || !document.createTextNode) return false;

  var abbreviations = document.getElementsByTagName("abbr");
  if (abbreviations.length < 1) return false;
  var defs = new Array();
  for (var i=0; i<abbreviations.length; i++) {
    var current_abbr = abbreviations[i];
    if (current_abbr.childNodes.length < 1) continue;
    var definition = current_abbr.getAttribute("title");
    var key = current_abbr.lastChild.nodeValue;
    defs[key] = definition;
  }
  var dlist = document.createElement("dl");
  for (key in defs) {
    var definition = defs[key];
    var dtitle = document.createElement("dt");
    var dtitle_text = document.createTextNode(key);
    dtitle.appendChild(dtitle_text);
    var ddesc = document.createElement("dd");
    var ddesc_text = document.createTextNode(definition);
    ddesc.appendChild(ddesc_text);
    dlist.appendChild(dtitle);
    dlist.appendChild(ddesc);
  }
  if (dlist.childNodes.length < 1) return false;
  var header = document.createElement("h3");
  var header_text = document.createTextNode("Abbreviations");
  header.appendChild(header_text);
  var container = document.getElementById("content");
  container.appendChild(header);
  container.appendChild(dlist);
}
```

```
addLoadEvent(stripeTables);
addLoadEvent(highlightRows);
addLoadEvent(displayAbbreviations);
```

这里的 highlightRows()函数有了一些小变化：它没有采用直接设置 CSS 样式属性的办法，而是利用来自 global.js 文件的 addClass()函数去添加一个取值为 hightlight 的 class 属性。当用户把鼠标指针悬停在某个表格行的上方时，这个表格行将获得一个新的 class 属性值，即 hightlight。在那之前，highlightRows()函数会把这个表格行的老 className 属性值提取到一个名为 oldclassName 的变量，等用户把鼠标指针移开这个表格行时，这个函数会把 className 属性重新设置为 oldclassName 变量的值。

在 layout.css 文件里增加一些针对“缩略词语表”的样式声明：

```
dl {
  overflow: hidden;
}
dt {
  float: left;
}
dd {
  float: left;
}
```

在 typography.css 文件里也相应地增加一些样式声明：

```
dt {
  margin-right: 1em;
}
dd {
  margin-right: 3em;
}
```

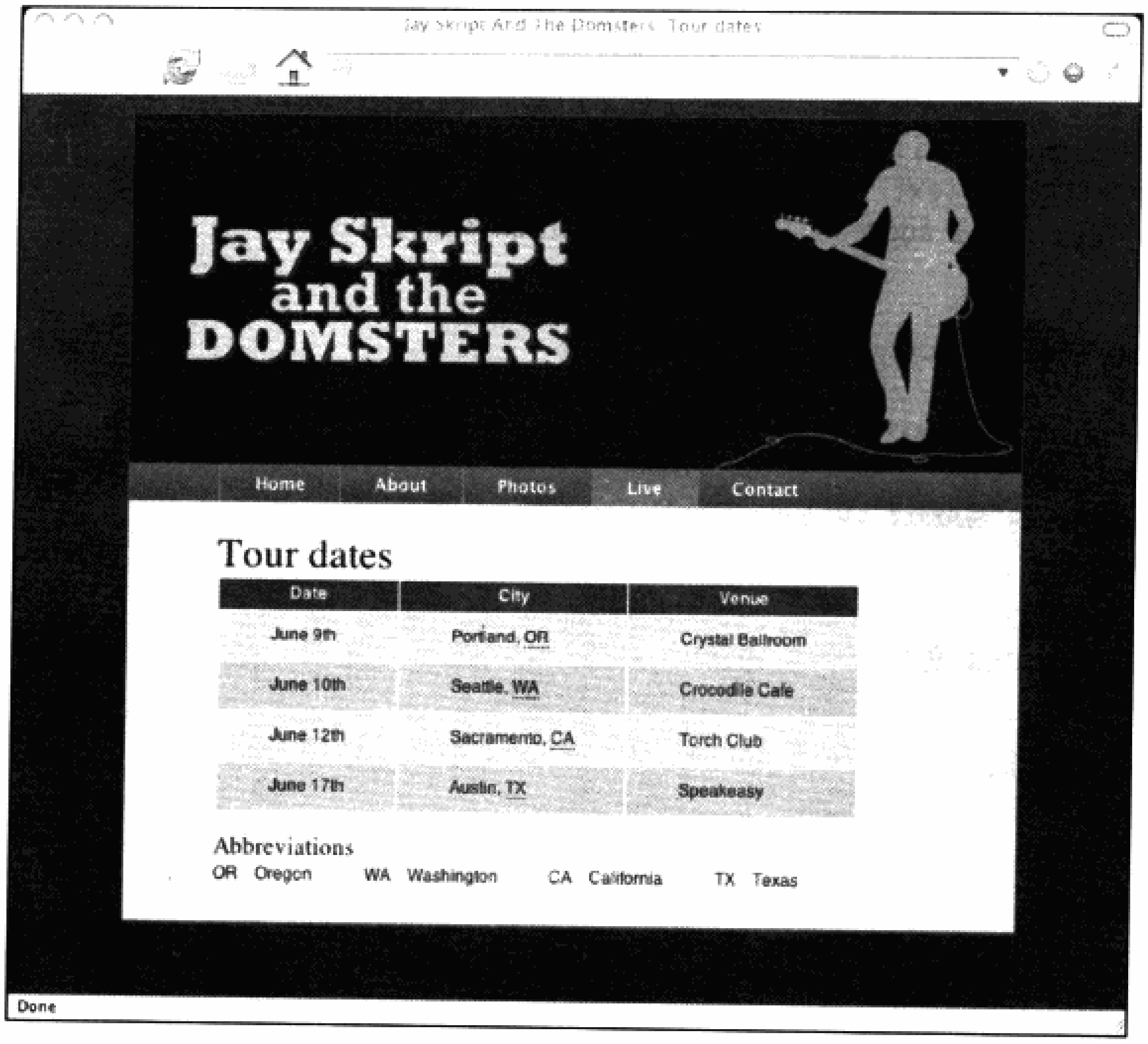
最后，在 color.css 文件里为 class 属性值是 odd 和 highlight 的元素增加一些颜色样式信息：

```
tr.odd td {
  color: #223;
  background-color: #ec8;
}
tr.highlight td {
  color: #223;
  background-color: #cba;
}
```

在 live.html 文件里插入一个如下所示的<script>标签来调用 live.js 文件：

```
<script type="text/javascript" src="scripts/live.js"></script>
```

把 photos.html 文件加载到 Web 浏览器里，就可以看到乐队演出日程表的改进效果了：这个表格中的每个偶数行现在都有了一个取值为“odd”的 class 属性。



11.6.6 改进表单

这个网站现在只剩下最后一个页面还未编写。这是一个很重要的页面——在网站上为访问者提供一些反馈渠道非常有必要。

差不多每个网站都会向访问者提供某种类型的联系信息，最起码会留下一个电子邮件地址。至于你正在创建的这个网站，则需创建一个联系表单。

联系表单需要用某种服务器端的技术来处理由用户通过表单输入和提交的数据。这种处理可以用 PHP、Perl、ASP 或者任何一种其他的服务器端程序设计语言来实现。不过，具体到这个例子，数据并不会被发送到服务器去——别忘了，这个乐队是虚构出来的。

首先，创建 contact.html 文件。它应该有着与 template.html 文件一样的基本结构，但在 id="content"的那个 div 元素里应该包含着如下所示的<form>标签：

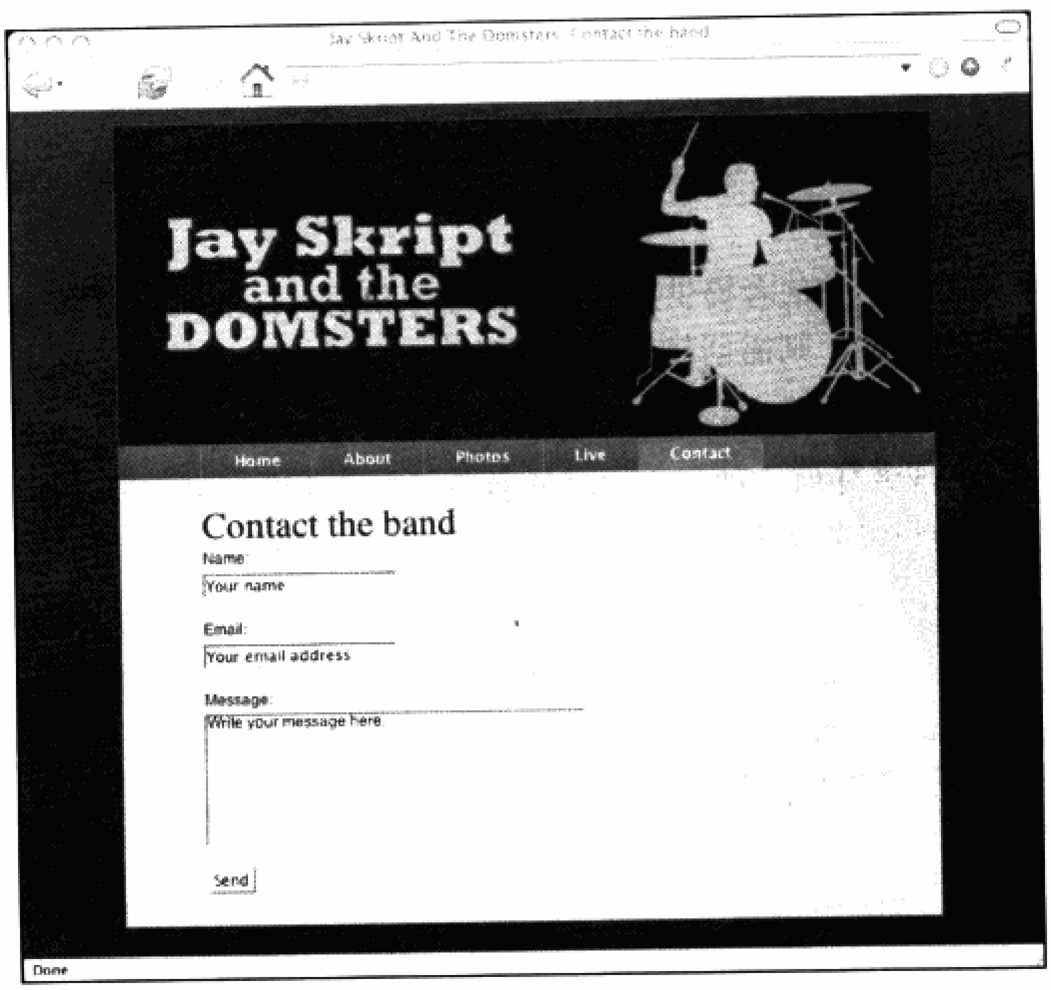
```
<h1>Contact the band</h1>
<form method="post" action="#">
  <fieldset>
    <p>
      <label for="name">Name:</label>
      <input type="text" id="name" name="name" value="Your name"
```

```
➥ class="required" />
    </p>
    <p>
      <label for="email">Email:</label>
      <input type="text" id="email" name="email"
➥ value="Your email address" class="email required" />
    </p>
    <p>
      <label for="message">Message:</label>
      <textarea cols="45" rows="7" id="message" name="message"
➥ class="required">Write your message here.</textarea>
    </p>
    <input type="submit" value="Send" />
  </fieldset>
</form>
```

然后，刷新 layout.css 文件：

```
label {
  display: block;
}
fieldset {
  border: 0;
}
```

下面是这个联系表单现在的样子。



1. <label>标签

这个联系表单有三个字段：name、email 和 message。每个字段都有一个相应的<label>标签。

label 元素特别有助于改善网页的可访问性。label 元素的 for 属性可以把一条文本与某个表单字段关联在一起。这对那些屏幕读取软件来说非常有价值。

label 元素对没有任何视力障碍的访问者也同样有价值。许多浏览器会为 label 元素创建这样一种默认行为：如果用户点击了某个 label 元素所包含的文本，与之相关的表单字段就将获得输入焦点，并等待用户输入数据。这个细节不太容易引起人们的注意，但确实会改善网站的可用性。但令人遗憾的是，并非所有的浏览器都实现了这种行为。

我们不能要求每种浏览器都实现了这种行为，但这并不影响我们把这种行为添加到我们的网页上。事实上，只需编写很少的几行 JavaScript 代码就可以做到这一点：

- 把文档里的所有 label 元素检索出来。
- 如果这是一个带有 for 属性的 label 元素，给它加上一个事件处理函数。
- 当这个 label 元素被点击时，提取出 for 属性的值。
- 这个值是某个 form 元素的 id。
- 检查这个 form 元素是否真的存在。
- 让这个 form 元素获得输入焦点。

把这个函数命名为 focusLabels。用 addLoadEvent()函数在页面加载时执行这个函数：

```
function focusLabels() {
  if (!document.getElementsByTagName) return false;
  var labels = document.getElementsByTagName("label");
  for (var i=0; i<labels.length; i++) {
    if (!labels[i].getAttribute("for")) continue;
    labels[i].onclick = function() {
      var id = this.getAttribute("for");
      if (!document.getElementById(id)) return false;
      var element = document.getElementById(id);
      element.focus();
    }
  }
}
addLoadEvent(focusLabels);
```

把以上代码保存为 contact.js 并把它存放到 scripts 文件夹。在 contact.html 文件的<head>部分插入一个如下所示的<scriptt>标签：

```
<script type="text/javascript" src="scripts/contact.js"></script>
```

把 contact.html 文件加载到 Web 浏览器里。点击某个 label 元素的标识文本，就可以看到与之相关的表单字段获得了输入焦点。根据你所使用的浏览器，这或许已经是它的默认行为。但

刚才做的这些事还是很有意义的，因为你可以确保这种行为在任何浏览器里都能实现。

2. 默认值

联系表单里的每个字段都有一些默认的“占位符”文本。name 字段里的默认文本是“your name”，email 字段里的默认文本是“your email”，等等。

从可访问性的角度看，这种做法非常有必要。W3C 在其 *Web Accessibility Initiative Guidelines*（关于如何提高 Web 内容的可访问性的指南）的 10.4 节里是这样说的：“如果你无法断定用户端的软件或代理能够对空白的控件做出正确的处理，就一定要在编辑框和文本输入区域里给出一些默认的、占位符性质的字符。[优先级 3]”。

因为历史的原因，有些浏览器在识别空白的表单字段时会出现种种问题。这会给那些习惯使用键盘浏览网页的用户带来许多困难。访问者甚至有可能根本无法利用 Tab 键到达或是离开空白的表单字段。

类似于<label>标签的情况，弥补这一缺陷可以给每个人带来好处。即便是对那些从不用键盘浏览网页的用户来说，表单字段里的默认文本也会非常有用，可以看到自己应该在每个表单字段里填写些什么是一种极大的便利。

在表单字段里放上一些默认的文本有这样一个美中不足：当用户打算在某个字段里输入内容时，他们必须先清除那些默认的占位文本才能开始输入自己的数据。这多少会给用户带来一些不便，会引起用户的反感。要是能让这种清除动作自动发生的话，就可以避免那些不便和反感了。

你猜怎么着，这还真的可以用 JavaScript 脚本实现。不过，这个任务用 DOM Core 的方法和属性来完成会相当复杂，你可以用 HTML-DOM 中最有用的对象之一——Form 对象来实现。

3. Form对象

我们知道，文档中的每个元素都是一个对象。每个元素都有诸如 nodeaName、nodeType 之类的 DOM 属性。

除了 DOM Core 提供的各种属性以外，有些元素还有其他一些属性。文档里的每个 form 元素都是一个 Form 类型的对象。每个 Form 对象都有一个 elements.length 属性。这个属性的值是包含在某给定 form 元素里的表单元素的总个数：

```
form.elements.length
```

这个值与 childNodes.length 是不同的，后者返回的是包含在某给定元素里的节点的总个数。Form 对象的 elements.length 属性只对表单元素——比如 input 元素、textarea 元素，等等进行统计。

同一表单里的所有表单字段构成了与这个表单相对应的那个 Form 对象的 elements 属性，这个属性其实是一个包含着所有表单元素的数组：

```
form.elements
```

同样地，这与 childNodes 属性也是有区别的。childNodes 属性也是一个数组，它包含着某给定元素的所有子节点。elements 数组将只返回 input 元素、select 元素、textarea 元素和其他表单字段。

elements 数组里的每个表单元素都有自己的一套属性。比如说，表单元素的 value 属性包含着该元素的当前值：

element.value

这相当于：

element.getAttribute("value")

你可以查询的另一个属性是表单元素的 defaultValue 属性，它包含着该表单元素的初始值：

element.defaultValue

如果你想知道某个表单字段的初始值，就应该去查询它的 defaultValue 属性。这正是你在即将编写的那个函数里所需要的东西。

contact.html 页面里的每个表单字段都有一个默认的初始值，它们可以让用户一目了然地知道自己应该在每个字段里输入的数据。不过，在开始输入之前必须先删除这些默认值，让用户以手动方式来做这种删除多少会给用户带来些不便。更好的解决方案是：当表单字段获得输入焦点时自动删除它的默认值，当用户在未输入任何东西的情况下离开表单字段时恢复它的默认值。

下面来编写一个名为 resetFields 的函数，这个函数只需要一个 Form 对象作为它仅有的参数：

- 遍历表单里的所有元素。
- 如果该元素是一个“提交”按钮，跳转到下一次循环的开始。
- 如果该元素没有默认值，跳转到下一次循环的开始。
- 否则，为“该元素获得输入焦点”事件增加一个事件处理函数：
- 把该元素的值设置为空。
- 再为“该元素失去输入焦点”事件增加一个事件处理函数：
- 如果该元素的值为空，把它改回该元素的默认值。

下面是 resetFields()函数的代码清单：

```
function resetFields(whichform) {
  for (var i=0; i<whichform.elements.length; i++) {
    var element = whichform.elements[i];
    if (element.type == "submit") continue;
    if (!element.defaultValue) continue;
    element.onfocus = function() {
      if (this.value == this.defaultValue) {
        this.value = "";
      }
    }
```

```
      element.onblur = function() {
        if (this.value == "") {
          this.value = this.defaultValue;
        }
      }
    }
  }
```

这个函数用到了两个事件处理函数。onfocus 事件将在用户用 Tab 键或鼠标选中某个元素时被触发；onblur 事件将在用户让输入焦点离开该元素时被触发。

把 resetFields()函数保存到 contact.js 文件里。你需要通过向 resetFields()函数传递一个 Form 对象来调用它。于是，再编写一个 prepareForms()函数，这个函数将遍历文档中的每个 Form 对象并把它们依次传递给 resetFields()函数：

```
function prepareForms() {
  for (var i=0; i<document.forms.length; i++) {
    var thisform = document.forms[i];
    resetFields(thisform);
    }
  }
}
```

用 addLoadEvent()函数调用 prepareForms()函数：

```
addLoadEvent(prepareForms);
```

重新加载 contact.html 页面以查看这个脚本的效果。

点击任意一个表单字段或与之对应的<label>标签文本，字段里的默认值将消失。如果你在未输入任何内容的情况下转到另一个字段，那个默认值将重新出现。如果你输入了一些内容，那个默认值将不重新出现。

- **表单数据的合法性检查**

你将在联系表单上完成的最后一项任务是 JavaScript 脚本的传统应用之一：对表单数据进行合法性检查。

在客户端对表单数据进行合法性检查是最古老的 JavaScript 应用之一。这种检查的原理很简单：在用户提交表单时，对他们提供的数据进行一些检查，如果用户提供的数据不符合要求（比如，用户没有填写必须填写字段、用户填写的数据有明显的错误，等等），就用一个 alter 对话框告诉用户哪些字段需要修改。

这听起来并不复杂，做起来通常也不困难。但如果用来检查表单数据的 JavaScript 代码编写得不够好，就会出现事与愿违的情况。如果 JavaScript 代码编写得很糟糕，用户甚至有可能永远也无法成功地提交那个表单。

在编写用来检查表单数据的 JavaScript 脚本时，必须要记住：

- 糟糕的表单检查还不如根本没有检查。
- 不要完全依赖 JavaScript，它代替不了服务器端的数据合法性检查。你已经用 JavaScript 对表单数据进行过检查，绝不意味着你在那些数据到达服务器时用不着再对它们进行检查。

知道了这些原则，你就应该把对表单数据的合法性检查弄得尽量简单一些。一项非常简单的测试是看用户是否真的提供了一个值。

下面是 isFilled()函数，它需要一个来自表单的元素作为它仅有的参数。如果该字段已被填写，这个函数将返回 true；如果用户没有填写那个字段，它将返回 false：

```
function isFilled(field) {
  if (field.value.length < 1 || field.value == field.defaultValue) {
    return false;
  } else {
    return true;
  }
}
```

通过检查 value 属性的 length 属性，你可以知道 value 属性的值是不是少于一个字符：如果是，这个函数将返回 false；如果不是，它将继续进行下一项检查。

通过比较 value 属性和 defautValue 属性，你可以知道用户是不是根本没有改动该字段里的默认文本：如果这两个值一样，这个函数将返回 false。

如果上述两项测试都通过了，isFilled()函数将返回 true。

下面是一个名为 isEmail 的类似函数，它负责检查某个表单字段的值像不像一个电子邮件地址：

```
 function isEmail(field) {
  if (field.value.indexOf("@") == -1 || field.value.indexOf(".") == -1)
{
    return false;
  } else {
    return true;
  }
}
```

这个函数利用 indexOf()方法进行了两项测试。indexOf()方法的基本用途是，找出一个字符串在另一个字符串里第一次出现的位置。如果找到了搜索字符串，它将返回那个位置；如果未找到，它将返回–1。

第一项测试是在表单字段的 value 属性值里寻找“@”字符。“@”是电子邮件地址里必须有的一个字符。如果没有找到“@”字符，isEmail()函数将返回 false。

第二项测试与第一项测试的原理一模一样，只不过这次测试是寻找“.”字符而已。如果未在表单字段的 value 属性值里找到“.”字符，这个函数将返回 false。

如果上述两项测试都通过了，isEmail()函数将返回 true。

isEmail()函数检查得并不细致——假的电子邮件地址或者是根本不是电子邮件地址的字符串完全有可能通过它的检查。不过，让这个函数太聪明了也不好。测试工作变得越复杂，出现“负误判”的概率也就越大。换句话说，如果对表单数据进行的检查太过复杂的话，合法数据遭到拒绝的概率就会增加。

现在有了两个函数：isFilled()和 isEmail()。你用不着对每个表单元素都进行这两项检查。你需要一种办法来区分“这个字段必须填写”和“这个字段应该是电子邮件地址”这两种情况。

在你的 XHTML 文档里，增加一个取值为 required 的 calss 属性：

```
<input type="text" id="name" name="name" value="Your name"
➥ class="required" />
```

还需要把“required”和一种取值为“email”的 class 组合在一起：

```
<input type="text" id="email" name="email" value="Your email address"
➥ class="email required" />
```

你可以在 CSS 文件里使用这些 class 属性值。如果你愿意，可以为 required 类别的字段设置一个更宽的边框或是另外一种背景颜色。

还可以在 JavaScript 脚本里使用这些 class 属性值。

下面来编写一个名为 validateForm 的函数，它需要一个 Form 对象作为它仅有的参数。

- 遍历表单里的 elements 数组。
- 如果在某个元素的 className 属性值里找到了字符串“required”，就把这个元素传递给 isFilled()函数。
- 如果 isFilled()函数的返回值是 false，显示一个 alert 对话框并让 validateForm()函数返回 false。
- 如果在某个元素的 className 属性值里找到了字符串“email”，就把这个元素传递给 isEmail()函数。
- 如果 isEmail()函数的返回值是 false，显示一个 alert 对话框并让 validateForm()函数返回 false。
- 否则，让 validateForm()函数返回 true。

下面是最终完成的 validateForm()函数的代码清单：

```
function validateForm(whichform) {
  for (var i=0; i<whichform.elements.length; i++) {
    var element = whichform.elements[i];
    if (element.className.indexOf("required") != -1) {
      if (!isFilled(element)) {
        alert("Please fill in the "+element.name+" field.");
        return false;
      }
```

```
      }
      if (element.className.indexOf("email") != -1) {
        if (!isEmail(element)) {
          alert
➡("The "+element.name+" field must be a valid email address.");
          return false;
        }
      }
    }
    return true;
  }
```

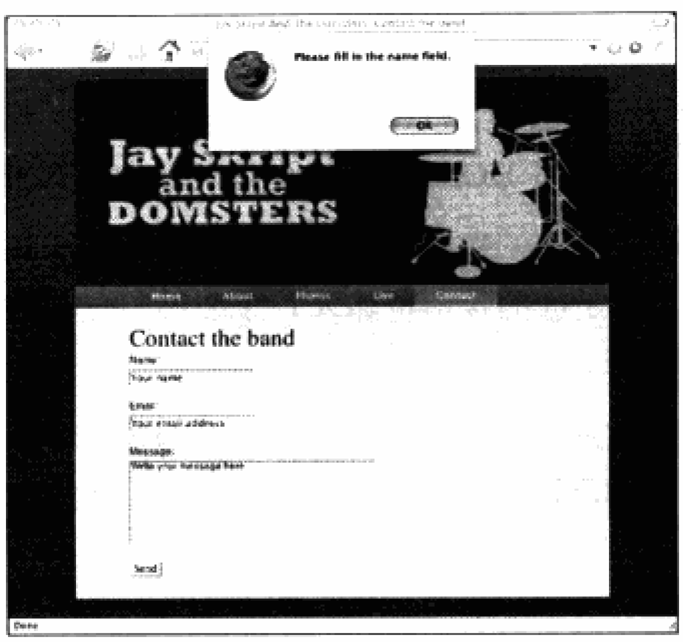
需要你安排的最后一件事是，把表单在它们被提交时传递给 validateForm()函数。请编写一个如下所示的 prepareForms()函数把上述行为添加到 onsubmit 事件处理函数上：

```
function prepareForms() {
  for (var i=0; i<document.forms.length; i++) {
    var thisform = document.forms[i];
    resetFields(thisform);
    thisform.onsubmit = function() {
      return validateForm(this);
    }
  }
}
```

用户提交一个表单的操作动作将触发一个 submit 事件，该事件将被 onsubmit 事件处理函数截获。这时，联系表单将被传递到 validateForm()函数。如果 validateForm()函数返回 true，表单就将被传递给服务器；如果 validateForm()函数返回 false，表单提交操作就将被取消。

把所有的表单检查函数保存到 contact.js 文件里。

在 Web 浏览器里刷新 contact.html 页面，然后用空白或默认值提交这个页面里的联系表单，你将看到一个 alter 对话框告诉你应该修改的第一样东西。



“Contact”页面就此完成，整个网站的设计和实现工作也就此结束了。

11.7 小结

“Jay Skript and the Domsters”乐队的网站已经可以发布到网上去了。你已经为这个乐队创建了一个有特色的网上形象。你用一些合法的、语义清晰的 XHTML 文档“包装”了这个网站的内容。还用一些外部 CSS 样式表为这个网站实现出了有特色的视觉效果。最后，利用 JavaScript 和 DOM 所提供的强大功能改善了这个网站的行为和可用性。

现在，唯一的遗憾是这个乐队是虚构出来的，但幸好这并不影响我们把最终的结果发布到网上。你们可以在网址 http://domscripting.com/domsters/index.html 处看到最终完成的网站。这个网站有以下几个特点：

- “Home”主页上有动画。
- “About”页面上的内容可以有选择地隐藏或显示。
- “Photos”页面上的图片有着与本书中的“JavaScript 美术馆”案例同样的浏览效果。
- “Live”页面里的演出日程表有着清晰易读的样式，那些样式是用 JavaScript 脚本实现的。
- “Contact”页面上的联系表单只有在用户填写了那几个“必须填写”的字段后才会被提交。

即使你把某一项或全部改进去掉，这个网站的浏览效果和运转情况也相当完美。那些 DOM 脚本并不是必不可少的，但有了它们，用户在访问这个网站时就会有更愉悦的体验。

11.8 下章简介

从某种意义上讲，你们已经学成出师了。

你们不仅学到了 DOM 脚本设计工作中的理论，还运用那些理论创建出了一个完整的网站。利用掌握的 DOM 方法和属性，你们已经可以编写出既实用又功能强大的 JavaScript 函数。

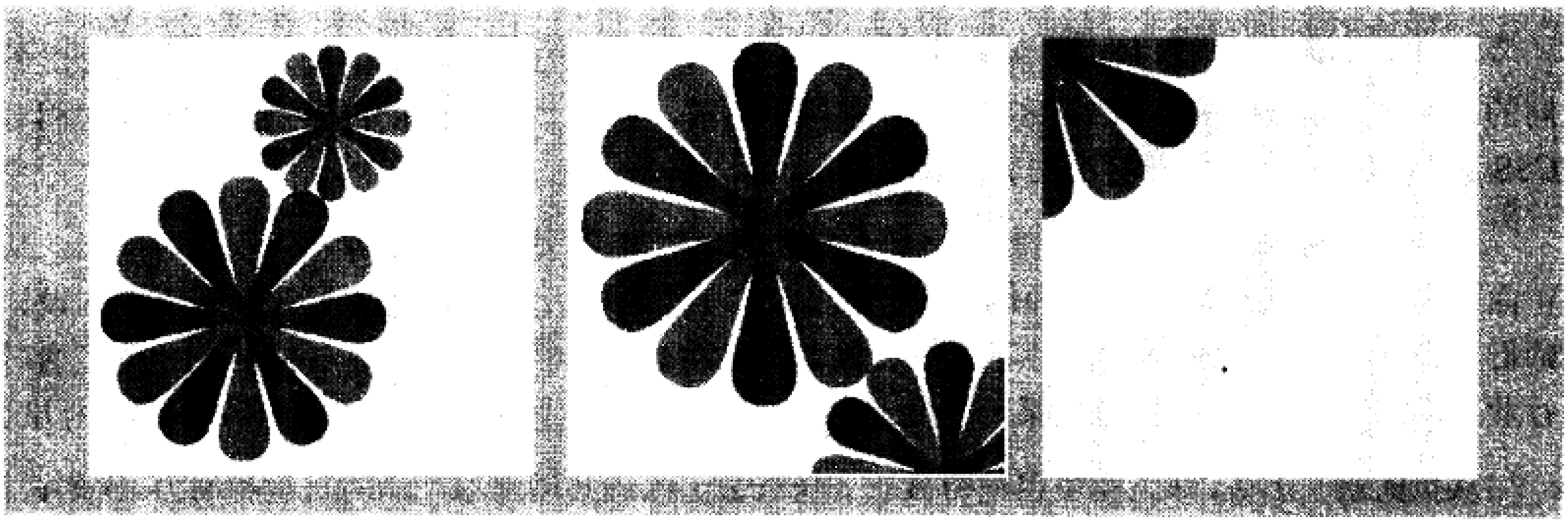
但从另一种意义上讲，你们才刚刚起步。

你们在这本书里看到的只是一部分 DOM 方法的一部分应用。那些 DOM 方法还有很多其他的用途在等待你们去发掘。还有许多 DOM 方法是我甚至还没来得及向你们介绍的。

要想编写出高质量的 DOM 脚本，知道多少 DOM 方法和属性并不是最重要的因素。编写出来的脚本是否足够健壮、是否有足够的预留退路才是重要的，而最重要的原则是必须把网页的结构层、表示层和行为层分离开来。

人们对 JavaScript 语言和 DOM 的应用一天比一天多。DOM 脚本编程技术把网站和网页的设计工作推向了一个更激动人心的高度。在本书的最后一章里，我将尝试着对未来做一些预测。

第 12 章 展望 DOM 脚本编程技术



本章内容

- ❑ Web 的现状
- ❑ Ajax 技术和 XMLHttpRequest 对象
- ❑ Web 上的应用软件

在本书即将结束时，我想对 JavaScript 语言和 DOM 的应用现状做一个总结。这些现有的应用为我们描绘了一个激动人心的未来。

在这本书的开始，我对 JavaScript 语言的历史和标准化 DOM 的兴起做了一个简单的回顾。回顾过去是比较容易的——作为事后诸葛，人人都能做到出言必中。但展望未来就不那么简单了，可我仍想这样去尝试一下。我的预测有可能全部落空，我也很可能会被人们说成是又一个无知妄言的典型。有了这个心理准备，我想请大家先和我一起去看看 Web 的现状。

12.1 Web 的现状

到底什么是 Web，对这个问题真是仁者见仁，智者见智。在某些人看来，Web 是电子商务；在另一些人看来，Web 是一个巨大的美术馆；还有许多人把 Web 看做是一个结交朋友和巩固友谊的虚拟社区。

这些都没错，但又都不全面。想通过探究 Web 上的内容去发现其本质是徒劳的。

不过，在一个更基础的层面上，万维网还是相当容易描述的：Web是所有网页构成的一个集合体。人们——通常是一些 Web 设计师创建了这些网页，把它们组织成各种各样的网站。在绝大多数场合，人们使用 Web 浏览器去访问这些网站。

这是一个相当简化的描述，但它确实可以帮助我们了解网站是如何制作和显示的。

12.1.1 Web浏览器

与 20 世纪 90 年代中期相比，现如今的 Web 浏览器市场正处于一种令人欣慰的平稳态势之中。

在过去，Web 浏览器在底层技术方面就有很大的分歧，来自不同厂商的浏览器往往会提供不同的（往往还是专有的）DOM 和 HTML 扩展。目前，Web 浏览器之间的区别只体现在它们对 CSS 和 W3C DOM 等行业标准的支持程度上。

在打赢了与 Netscape 公司的战争之后，微软公司的 IE 浏览器已经主宰市场很长时间了。从正面意义上讲，这对 Web 设计师们是一个好消息。一个稳定发展的浏览器市场意味着 Web 设计师们在设计和发布 Web 内容时，不再需要像以前那样不得不考虑许多不确定因素。IE 浏览器对行业标准的支持相当完善，这在某种程度上也促使了 Web 设计师们去学习和遵守各项 Web 标准。

从负面意义上讲，缺乏竞争也使得微软公司失去了改进其浏览器产品的兴趣和动力。比如说，IE 浏览器对 CSS 的支持其实只要再增加不多的几项就可以得到巨大的改善，就可以让它更符合有关的标准，但微软迟迟没有在这方面采取行动。

过去的几年中，Web 设计师们不得不一再容忍这种迟钝。这是一种让人着急的局面：主宰着市场的 Web 浏览器对标准的支持很不错，但它本来可以做得更好；浏览器市场在相当长的时期里很稳定，但谈不上完美。

现在，终于发生了一些打破这种僵局的事情。

Firefox浏览器

由 Mozilla 推出的 Firefox 浏览器（`http://www.mozilla.org/products/firefox/`）使用了开放源代码的 Gecko 呈现引擎。Gecko 对 Web 标准的支持非常完善，以它为基础的 Firefox 浏览器自然也不差。更给人以信心的是，对 Gecko 的改进一直在有条不紊地进行着。

大多数人并不关心 Web 标准。我敢保证，在挑选浏览器时，根据它们对 W3C 推荐的标准的支持程度去做出决定的人肯定是极少的。但人们肯定会关心速度、易用性和信息安全等问题。事实上，正是因为这些显而易见的理由，才会有越来越多的人把自己的 IE 浏览器换成了 Firefox 浏览器。

浏览器市场多年来的平衡局面被打破了。微软公司已经开始注意这个动向。停滞多年后，IE 浏览器的研发工作终于重新启动了。



另一场浏览器大战的战场已经形成，但我认为这次的事态发展会与以往不同：在这第二次浏览器大战中，参战各方都会使用正大光明的武器，在第一次浏览器大战中导致 Web 开发环境污染多年的“脏弹”应该不会重现战场。

我希望，发生在 IE、Firefox 以及其他主动或被迫加入战团的浏览器产品之间的战争，能够像任何其他的市场份额争夺战那样给我们带来惊喜而不是灾难：有竞争力的产品层出不穷，而它们赖以打开和扩大市场份额的手段是提供优于竞争对手的功能。对大多数人来说，这将意味着更流畅的浏览体验、更好的垃圾内容阻断机制和更安全的信息共享。对 Web 设计师来说，这将意味着对 CSS2 和 DOM 等行业标准的更好支持。

12.1.2 Web 设计师

“Web 设计师”并没有一个正式的定义。它不仅对不同的人会有不同的含义，它在不同的时间也会有不同的含义。下面这个定义来自 Wikipedia（`http://en.wikipedia.org/wiki/Web_designer`）：

“从广义上讲，‘Web 设计’指的是网页、Web 站点或 Web 应用软件的设计结果或设计过程。从狭义方面看，这个术语通常特指使用图片、CSS 和 XHTML 进行的 Web 开发项目中与图形图像有关的方面。”

就在几年前，在“Web 设计”的定义里还找不到“Web 应用”的字眼。CSS 的兴起也只是一种相对较晚的现象。在 21 世纪刚开始时，CSS 还是一种绝大多数 Web 设计师都不甚了解的边缘

技术。但今天，它已经成为每个 Web 设计师都能娴熟运用的看家本领了。

有些 Web 设计师只用 CSS 去改变颜色和字体。另一些 Web 设计师则用 CSS 去完成一切事情，包括对页面元素的摆放位置做出安排和调整。后一种 Web 设计师目前还只是少数，但他们的队伍正在逐渐扩大。现在的问题已不再是要不要用 CSS 去安排页面的布局，而是在什么时候这样做。根据 Web 设计师的个人水平和经验，从利用表格元素安排页面的布局过渡到利用 CSS 完成这类任务会有一个或长或短的过程，但这种趋势本身看来已不可逆转。

基于 CSS 的设计正越来越多，这让我在欣慰之余也不免感到有些吃惊。在向 CSS 过渡的过程中，Web 设计师需要面对的最大障碍是变化莫测的浏览器支持问题。IE 浏览器对 CSS 不够完善的支持如今已经成为 CSS 技术进一步发展的绊脚石。

与 CSS 的情况相比，DOM 脚本编程技术就像是公园里的一条小路。这条路上有几块小石子——有些浏览器对 DOM 的支持还不够完善，但没有大的障碍——主流的浏览器都全面支持 DOM。

尽管如此，知道 CSS 的 Web 设计师在人数上还是要比知道 JavaScript 语言和 DOM 的要多得多。我认为这主要有以下几个方面的原因：

- JavaScript 语言在过去的名声不太好。滥用 JavaScript 技术而实现出来的一些“改进”给人们留下了非常不好的印象。
- 因为缺乏预见性，许多早期的 JavaScript 脚本都存在着可访问性问题。那些问题使得人们产生了这样一种误解：JavaScript 语言是一种在可访问性方面有着先天不足的技术。
- 有相当一部分人至今仍认为 DOM 在不同的浏览器里有着不同的实现。这在 DHTML 技术标准之争和浏览器大战的黑暗时期的确如此，但如今早已不是这样了。
- JavaScript 是一种程序设计语言。许多 Web 设计师已经用惯了可视化编程工具，对亲自动手编写代码产生了抵触、甚至是恐惧的心理。

总之，DOM 脚本编程技术之所以未被人们广泛接受，最大的障碍只是一个公共关系问题，而它正在逐步地——虽然有些缓慢，得到解决。已经有越来越多的 Web 设计师开始认识到他们的知识存在着缺陷，并开始弥补自己在这方面的不足。

12.1.3　三条腿的凳子

有许多 Web 设计师都有为纸质印刷品设计版面的经验。事实上，设计网站时，把网页当做纸质印刷品来对待是最容易想到的思路，除了把白纸换成了屏幕，其他做法没什么两样。这种做法注定会失败。Web 是一种不同的媒体。在设计纸质印刷品时，必须把原始内容和版面设计交织在一起才能看到最终的印刷效果，但在设计网页时，只有将其分离开来才能获得最佳的结果。

最准确的网页设计思路是把网页分成三个层次，即：

(1) 结构层
(2) 表示层

(3) 行为层

这些层次中的每个都需要用一种不同的技术来实现，它们是：

(1)（(X)HTML [可扩展]超文本标记语言）
(2) CSS（层叠样式表）
(3) JavaScript 语言和 DOM（文档对象模型）

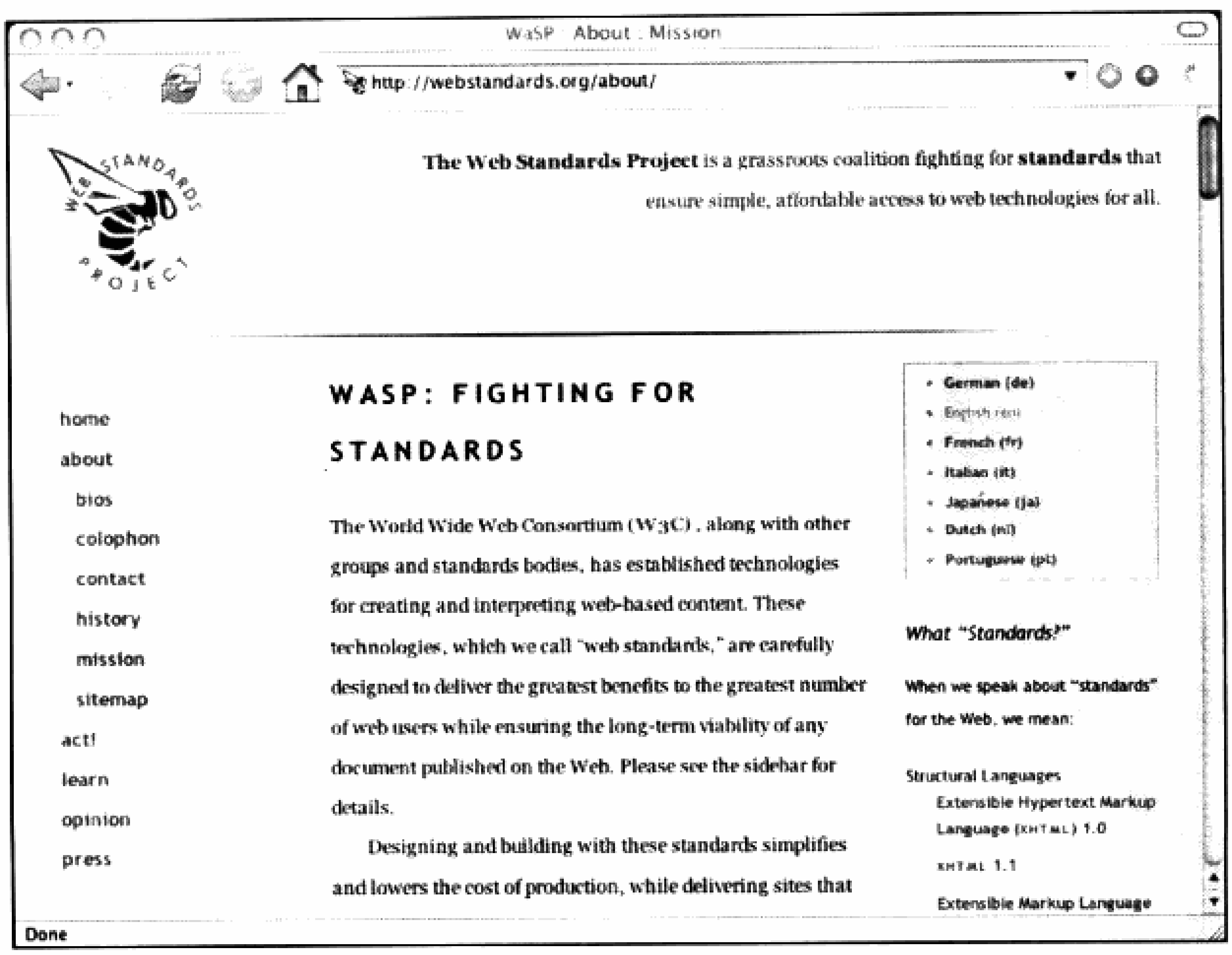
这三种技术就像是一个凳子的三条腿。要想成为一名技能全面的 Web 设计师，就必须熟练掌握这三种技术，并知道每种技术最适用于哪一类问题。目前，DOM 脚本编程技术可以说是那个凳子最短的一条腿。

许多 Web 设计师正开始认识到自己在知识和技能方面的缺陷。他们已经掌握了网页的结构和样式，现在正在努力学习如何去掌握网页的行为。

对这些 Web 设计师来说，学习一种像 JavaScript 这样的程序设计语言并不是件易事。但幸运的是，助手就在他们身边。

DOM Scripting 任务组

2005 年 7 月，由 Web 标准计划发起的 DOM Scripting 任务组（http://domscripting.webstandards.org/）正式宣告成立。这个工作组的主要任务是鼓励和帮助人们编写出遵守有关标准、分离 JavaScript、有足够预留退路的 DOM 脚本。

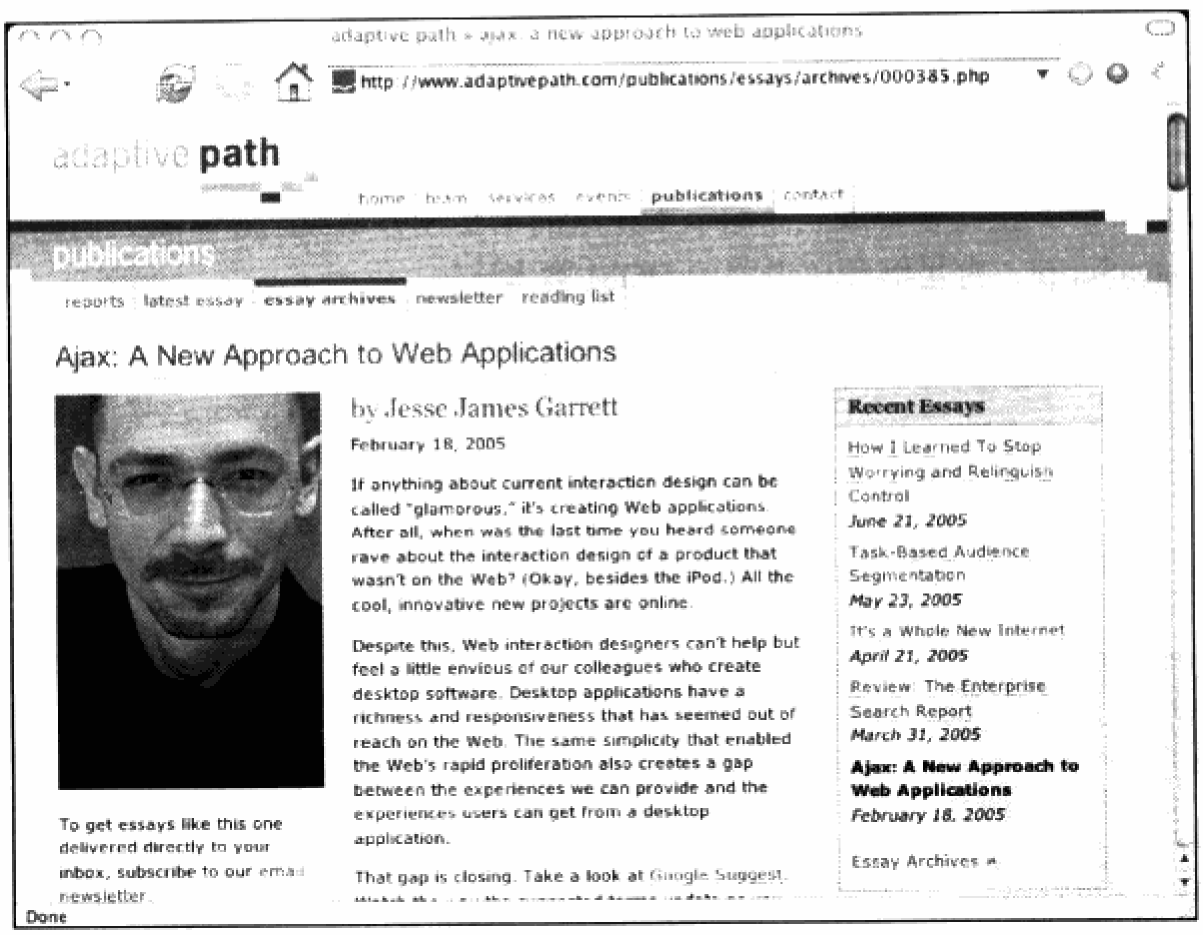


这本书只是一个更大规模的 DOM 最佳编程经验推广活动的组成部分之一。为了帮助那些希望提高自己知识和技能水平的 Web 设计师，这个任务组的各位成员定期推出有针对性的文章和教程。

我相信，DOM Scripting 任务组会受到 Web 设计师们越来越热烈的欢迎。有些已经发生的事情正在扭转人们对 JavaScript 语言的不良印象，Ajax 技术就是其中之一。

12.2　Ajax 技术

2005 年 2 月 18 日，Adaptive Path 公司的 Jess James Garrett 发表了一篇名为 *Ajax: A New Approach to Web Applications*（Ajax 技术：开发 Web 应用的新思路）的文章（详见 `http://www.adaptivepath.com/publications/essays/archives/000385.php`）。在那篇文章里，Ajax 既不是指那一位古希腊神话里的传奇英雄，也不是一种家用清洁产品，更不是科幻电影里的太空飞船。那篇文章的作者用 Ajax 来称呼一种革命性的 JavaScript 编程思路和方法。



Ajax 技术说的是把 JavaScript、CSS、DOM 和(X)HTML 结合起来的一种新用法。这种结合并不是新概念——事实上，DHTML 技术早就把这几样东西结合在一起了。Ajax 技术的独到之处是它在服务器端使用了异步（asynchronous）处理技术。

我们知道，Web 应用软件在运行时需要刷新大量的页面。用户在页面上做出一项选择或是输入一些数据，客户机把这些信息发送给服务器，服务器根据用户的操作动作返回一个新的页面。

即使用户只对服务器做了一次极其简单的查询，服务器也需要返回一个全新的页面。

就拿登录页面来说。除了登录表单，登录页面通常还会有网站的品牌信息、导航条和版权声明等。对于传统的登录页面，在某个用户提交了登录表单后，服务器将比较用户在登录表单里输入的数据和保存在某个数据库服务器里的登录信息。如果用户输入的数据不正确，服务器就把同样的登录页面再次发送给这位用户。在这个“新”登录页面里，网站的品牌信息、导航条和版权声明等与前一个登录页面一模一样，唯一的区别是“新”登录页面里多了一条告诉用户“本次登录失败”的消息。也就是说，虽然页面上只有一个很小的部分需要刷新，但实际刷新的却是整个页面。用户每发出一个请求，整个页面就会被全部刷新，页面加载与用户请求是同步的。

如果登录页面使用了 Ajax 技术，“新”页面上就只有登录部分会发生变化，网站的品牌信息、导航条和版权声明等都将是前一个页面留下来的遗产。在用户填写完登录表单并按下“提交”按钮之后，如果登录没有成功，出错消息将出现在已经被加载到浏览器里的那个“老”登录页面上。

传统的登录页面与 Ajax 版本的区别是：后一种情况里的服务器端处理是异步发生的，用户发出的每一个请求不见得都会导致整个页面被全部刷新一次，服务器可以在后台对请求进行处理。

Derek Powazek 为此给出了一个非常形象的比喻：传统的 Web 技术与 Ajax 技术之间的区别就像是电子邮件与即时通信那样。

客户端处理与服务器端处理一直有着非常明显的界线。在客户端，或者说在浏览器里，JavaScript 可以对当前页面的内容进行处理。一旦需要进行服务器端处理，就会有一个请求被发送到服务器上的某个程序，而位于服务器端的程序可以用 PHP、ASP、Perl、Coldfusion 或任何一种其他的服务器端程序设计语言来编写。过去，服务器向客户端返回响应的唯一办法是提供一个全新的页面。

传统情况下，每当客户（Web 浏览器）需要一些来自服务器的内容时，就会有一个请求从客户端发往服务器，对这个请求的响应又会从服务器返回给客户。

Ajax 技术等于是在客户和服务器之间安插了一个中转站：JavaScript 脚本先把请求从客户端发送给这个中转站，再由这个中转站把请求转发给服务器；服务器先把响应也发送给这个中转站，再由这个中转站把响应转发给客户端的 JavaScript 脚本处理。

我们刚才所说的“中转站”就是 XMLHttpRequest 对象。

12.2.1 XMLHttpRequest 对象

不要浪费时间和精力到 W3C DOM 的技术标准里去寻找 XMLHttpRequest 对象，因为你们在那里根本找不到它——尽管这个对象在现代的 Web 浏览器当中已经获得了广泛的支持，但它还未成为一项标准。

令人遗憾的是，不同的浏览器在 XMLHttpRequest 对象的具体实现和具体使用方面还没有统一的标准化做法。如果你想使用 XMLHttpRequest 对象，则需要在你的代码安排很多的测试和分支，就像你在与非标准化的 JavaScript 功能打交道时那样。

微软公司率先（作为它们的一个专利 ActiveX 对象）实现了 XMLHTTP 对象。下面是为 IE 浏览器创建一个新“中转站”（用术语来说，就是 XMLHTTP 对象的一个实例）的常见做法：

```
var waystation = new ActiveXObject("Microsoft.XMLHTTP");
```

为其他的浏览器创建一个新“中转站”需要使用 XMLHttpRequest 对象：

```
var waystation = new XMLHttpRequest();
```

如果想让自己编写出来的代码同时适用于这两种实现，就需要编写一个类似于下面这个 getHTTPObject()函数的东西去创建一个正确的对象实例：

```
function getHTTPObject() {
  if (window.ActiveXObject) {
    var waystation = new ActiveXObject("Microsoft.XMLHTTP");
  } else if (window.XMLHttpRequest) {
    var waystation = new XMLHttpRequest();
  } else {
    var waystation = false;
  }
  return waystation;
}
```

这只是一个非常简单的例子。在实际工作中，你往往需要编写一些更复杂的测试和分支。

getHTTPObject()函数将返回一个指向新创建的 XMLHttpRequest 对象的引用指针，你通常需要把它赋值给一个变量，如下所示：

```
request = getHTTPObject();
```

这个对象有一系列方法，其中最有用的是 open()方法，它将把这个对象与服务器上的某个文件关联起来。还可以在调用 open()方法时指定一种我们打算使用的 HTTP 请求类型：GET、POST 或 SEND。这个方法的第三个参数决定着是否要按异步方式来处理这个请求。

下面这条 JavaScript 脚本里的语句向 example.txt 文件发出了一个 GET 请求，然后服务器将到这个 JavaScript 脚本所在的子目录里去寻找 example.txt 文件：

```
request.open( "GET", "example.txt", true );
```

你还需要指定 XMLHttpRequest 对象接收到来自服务器的一个响应时采取的行动。你可以用这个对象的 onreadystatechange 属性来做这件事情。这是一个将在服务器把响应返回给 XMLHttpRequest 对象时被触发的事件处理函数。

下面这条语句将在 onreadystatechange 事件被触发时调用 dosomething 函数：

```
request.onreadystatechange = doSomething;
```

指定了这个对象发送请求的目的地和它接收到响应时将采取的行动，你就可以用 send()方法来启动这一系列动作了：

```
request.send(null);
```

下面是这一过程的完整步骤：

```
request = getHTTPObject();
request.open( "GET", "example.txt", true );
request.onreadystatechange = doSomething;
request.send(null);
```

当然，为了对来自服务器的响应做出处理，还需要把 dosomething()函数实实在在地编写出来。

当服务器把一个响应返回给 XMLHttpRequest 对象时，这个对象有几个属性将从“不可用”变成“可用”。readyState 属性包含着一个数值，这个数值将随着服务器对相关请求的处理进度而变化：它有 5 种可取值：

0 尚未初始化
1 正在加载
2 加载完毕
3 正在处理
4 处理完毕

一旦 readyState 属性的值变成了 4，就可以对从服务器返回的响应数据进行访问了。

从服务器返回的响应数据保存在 responseText 属性里，那是一个字符串值。如果响应数据是以 Content-Type 类型“text/xml”返回的，你还可以访问 responseXML 属性。responseXML 属性的值是一个 DocumentFragment 对象，所以你可以立刻使用各有关 DOM 方法去处理它——Ajax 技术中的 XML 成分就源自这里。

在下例中，dosomething()函数将在 readyState 属性的值变成 4 之后把整个 responseText 属性输出到一个 alert 对话框里：

```
function doSomething() {
  if (request.readyState == 4) {
    alert(request.responseText);
  }
}
```

如果被请求的 example.txt 文件的内容是“Hello world”，dosomething()函数就将把它显示在 alert 对话框里。

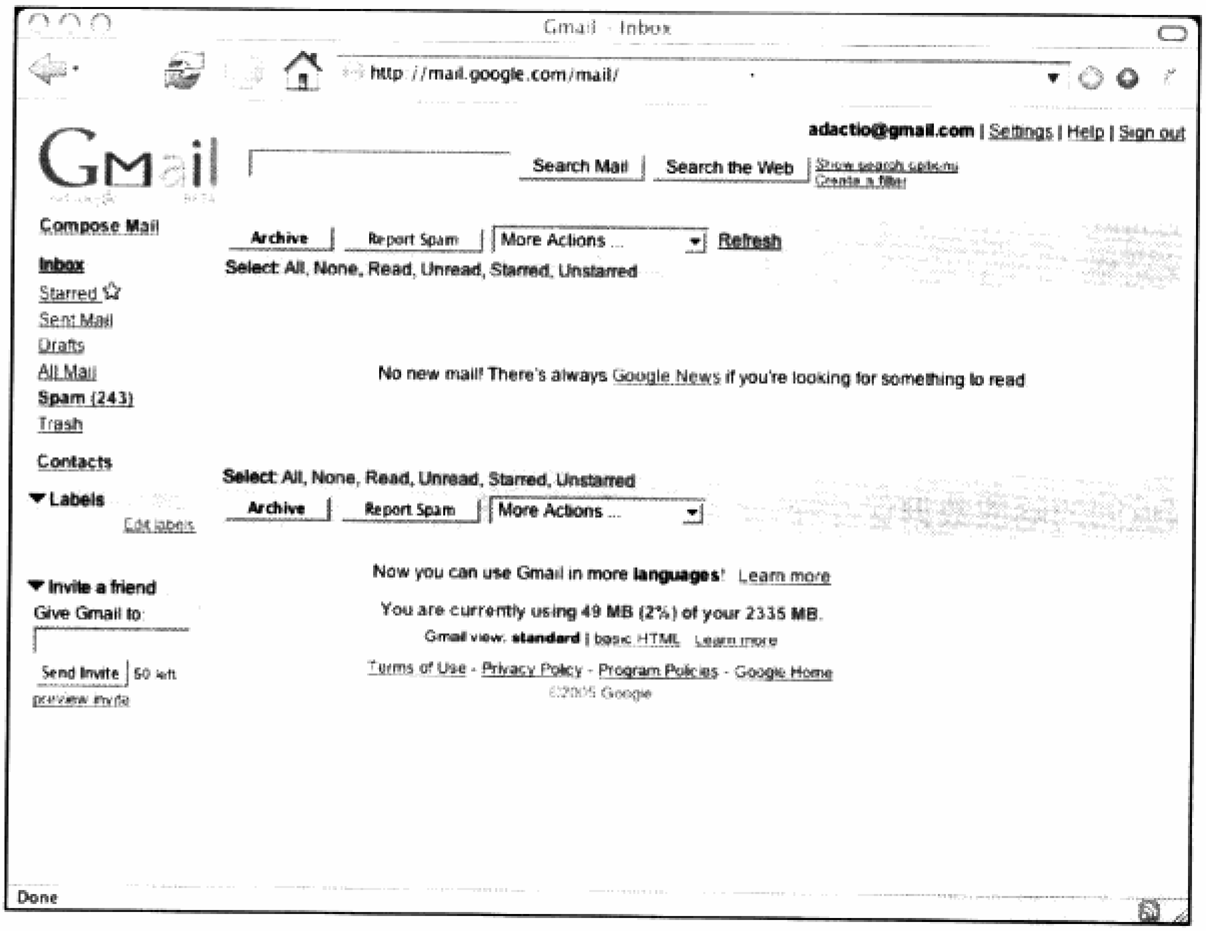
上面给出的示例非常简单。但是我相信，只要发挥一下你们的想像力，就可以为它想到许多令人叹为观止的用途来。

12.2.2　Ajax 技术的爆发

因为 Ajax 技术只是在最近才引起了人们的广泛关注，所以如果你把它误认为是一种新技术也无可厚非。事实上，XMLHttpRequest 对象已经面世很多年了。就技术而论，JavaScript 脚本以异步方式与服务器进行交互并不新鲜，而 Ajax 技术中的其他成分——CSS、(X)HTML 和 DOM 也都早已为人们所熟悉。

不过，人们对 Ajax 技术的热情直到 2005 年才被激发出来。这会不会是因为这种“新”技术有了一个像 Ajax 这样的好名字才使得它突然变得炙手可热呢？

其实不然，人们对采用这种“新”技术去创建网站的兴趣早在 Jess James Garrett 创造出 Ajax 这个名字之前就已经在不断升温了，而引爆人们热情的最后一根导火索是 Google 推出的新型 Web 应用。



在 Google 刚推出基于 Web 的电子邮件服务 Gmail（http://mail.google.com）时，最吸引人们的地方是它那超大的邮箱容量。但开始使用这项服务之后，其他一些功能开始成为人们关心的焦点。Gmail 能够在后台透明地提供拼写检查、电子邮件地址的自动补足以及其他一些任务。这项 Web 应用的前端以异步方式向服务器发送数据及从服务器接收数据。

到了 Google Suggest（http://www.google.com/webhp?complete=1）的 beta 测试版推出时，人们对 XMLHttpRequest 对象的兴趣被进一步地激发了出来。当用户开始在搜索字段里敲入短语时，这个字段下方的提示框会列出一些现成的短语供用户选择。用户每敲入一个字母，服务器就

会提供一组新的候选短语。因为客户/服务器之间的交互是以异步方式进行的，所以这项功能的效率非常高，用户几乎是立刻就能看到最新的提示。如果 Google Suggest 的工作原理是用户每敲入一个字母就重新加载一次整个页面的话，它的用处肯定会大打折扣。

有了 Google 提供的活生生事例，又有了 Jess James Garrett 想出的响当当名字，Ajax 技术就这样爆发了。

Ajax技术面临的挑战

Ajax 技术肯定会大行其道，这一点我敢保证。我认为，Ajax 技术将给我们带来许多好处。因为它能够在无需刷新整个页面的情况下对用户的操作动作做出非常迅速的响应，所以 Ajax 技术可以显著改善网站的可用性。但与此同时，这项“新”技术也面临着一些挑战。

Ajax 技术的特点之一是可以减少各有关页面的加载次数，这种“偷工减料”的行为难免会干扰到 Web 浏览器的某些操作——比如“后退”按钮或“书签”功能。

Ajax 技术的另一个特点是只刷新有关页面的一部分。从理论上讲，用户采取的每个行动都应该有一个明确和清晰的结果。因此，在采用 Ajax 技术之后，Web 设计师必须解决好在两个时刻提供有意义的提示信息的挑战：其一是在用户向服务器发出一个请求时；其二是在服务器向用户返回一个响应时。

Ajax 技术无疑是一种功能非常强大的工具，但与其他任何一种功能强大的工具一样，它也有可能被人们滥用。眼下正是 JavaScript 语言开始被人们广泛接受的关键时期，如果现在不警惕 Ajax 技术被滥用的苗头，就有可能让 JavaScript 语言因 Ajax 技术而再次背上“这种技术难以使用，基于这种技术的页面难以访问”的坏名声。

开发出一个成功的 Ajax 应用的关键是，应该把这种技术当做一种“调味品”而不是“必需品”来对待——就像 JavaScript 脚本应该用来充实网页而不是用来构造网页那样。换句话说，我们应该在基于 Ajax 技术的 JavaScript 脚本里预留出足够的退路。

12.2.3 循序渐进：如何运用 Ajax 技术

Ajax 技术能够让网页对用户的操作动作做出快速、透明的响应，这一特点使得基于 Ajax 技术的网站更像是一些桌面软件而不是传统的 Web 站点。说 Ajax 网站像是桌面软件没有错，但这绝不等于说 Ajax 网站就是桌面软件。在网站的设计和实现过程中，千万不要因为有了 Ajax 技术而忽视与网站的可用性和可访问性有关的基本原则。

令人遗憾的是，有些基于 Ajax 技术的网站已经把 JavaScript 当做访问该网站内容的先决条件了，而它们为此给出的理由竟然是：因为网站提供的功能太过丰富，所以无法再为网页预留出足够的退路。

我绝不认同这种说法！事实上，凭着我对 Ajax 技术内涵的了解，我坚信任何一种 Ajax 应用

都至少会有一种无需依赖于 Ajax 技术的解决方案，而这又完全取决于你打算如何运用 Ajax 技术。

如果从项目开发之初就一直用 Ajax 技术来实现其核心部分的功能，等到了后面的阶段再想把它分离出去，以获得一个不依赖于 Ajax 技术的版本将会非常困难。反之，如果先按照传统的网页刷新思路，把应用项目开发出来，再通过拦截将被发送给服务器的请求，并安排它们经由 XMLHttpRequest 对象转发的办法用 Ajax 技术去进一步充实已有的框架，结果将大不一样：网站不需要依赖于 Ajax 技术，Ajax 功能只是一个能够正常运转的网站的附加层次。

听起来很耳熟？没错，这与本书一直在强调的循序渐进的开发原则没有任何区别。

从一开始就把 Ajax 技术集成到 Web 应用的核心部分，就如同为了让一个链接去触发一个操作动作而在 HTML 代码里使用"javascript:"伪协议那样是不好的编程习惯。先创建一个普通的链接再去拦截它的默认行为才是正确的做法。总之，创建一个 Ajax 网站的最好办法是，先把它创建为一个普通的网站，再利用 Ajax 技术去充实它——我把这种思路和实践称为"**Hijax 技术**"。

Hijax

如果说 Ajax 技术的成功给我们以什么启示的话，"简短时髦的名字有助于技术的推广"绝对应该是其中之一。与"DOM、CSS 和 XTML 加上 XMLHttpRequest 对象"相比，"**Ajax**"无疑更琅琅上口。因此，我决定把"按照循序渐进的原则使用 Ajax 技术"简称为"**Hijax 技术**"。

Ajax 技术的威力来自服务器：几乎所有的实际操作都是由一种服务器端语言来完成的，XMLHttpRequest 对象只是一个负责在浏览器和服务器之间转发各种请求和响应的中转站。如果去掉它，发出请求和接收响应的工作应该仍能正常进行，只是等待的时间将会稍微长一些。

回到刚才所说的登录页面的例子。创建这种页面最简单的办法是按照久经时间考验的套路去做：让表单把整个页面发送给服务器，再由服务器返回一个包含着反馈信息的新页面。所有的处理工作全部在服务器上完成，在服务器上对用户输入的数据和保存在某个数据库里的数据进行对比，以找到一个匹配。

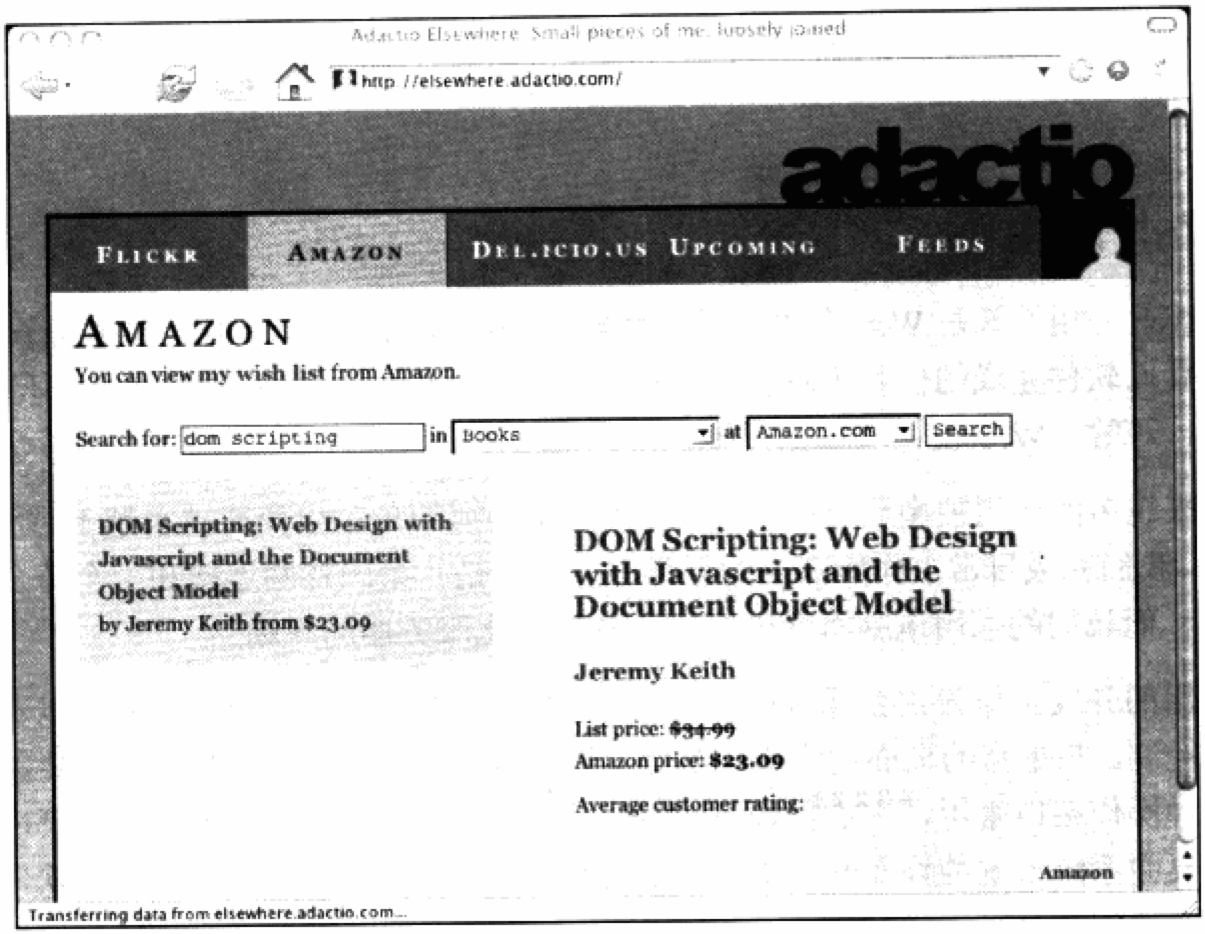
要想把 Ajax 技术应用到登录页面上，就必须把表单提交动作拦截下来并代之以经由 XMLHttpRequest 对象转发。表单提交动作将触发一个 submit 事件。如果用 onsubmit 事件处理函数把这个事件拦截下来，就可以把这一事件的默认行为（提交整个页面）撤消并替换为一个新的动作：把数据经由 XMLHttpRequest 对象传递给服务器。

一旦拦截到了登录表单，登录过程对用户来说就变得更方便了。响应时间缩短了，页面刷新次数减少了。最重要的是，即使用户没有激活 JavaScript，该项应用也可以正常工作。完成登录所需要的时间会有所增加，用户的体验会不那么完美，但因为登录处理工作是在服务器上完成的，所以没有理由让用户来承担责任。

基于 Ajax 技术的应用对服务器端处理的依赖远大于它们对客户端处理的依赖，这意味着它们肯定都有足够的预留退路。

应该承认，有些 Ajax 应用在“退化”到只能通过刷新整个页面来实现其功能的时候，会慢得让人难以忍受。可不管怎么说，再慢的浏览体验也要好过根本没有体验，对吧？

为了证明我在这里提出的Hijax理念是行得通的，我已经在网址http://elsewhere.adactio.com/处为大家准备了一个 Ajax 应用。



即使禁用了 JavaScript，你们也能看到这个网站上的所有内容。

12.2.4 Ajax 技术的未来

我相信，Ajax 技术肯定会大行其道。但就我个人而言，我更希望看到 Ajax 技术以及其他 DOM 程序设计技术能够被人们正确地运用：作为进一步提升用户体验的“调味品”而不是用户获得这种体验的“必需品”。我更希望看到人们把 Ajax 技术应用在诸如反馈表单和购物车之类的页面元素上。

在我看来，Hijax 技术是运用 Ajax 技术最简单的办法。但令人遗憾的是，有许多广受欢迎的 Ajax 应用并没有遵循这一理念。Gmail 和 Google Maps 都是从一开始就把 Ajax 技术机密地集成在了它们的核心功能里，这就使得它们很难再“退化”为一种无须依赖于 Ajax 技术的应用。

如果 Google 公司当初在开发 Google Maps 时能像其他的地图网站那样，把 Ajax 功能应用在 Google Maps 核心功能的外层并不困难。只可惜事实并非如此，而现在要想为 Google Maps 提供一个不依赖于 Ajax 技术的版本实在是太难了。

不过，Google Suggest 倒是一个循序渐进地运用 Ajax 技术的典型事例：它的核心功能是在 Web 上搜索一条词语；如果用户激活了 JavaScript，Ajax 技术提供的搜索词建议将给用户带来很大方便，但即使用户禁用了 JavaScript，其核心功能也能够照常使用。

Ajax 技术无疑是一项很了不起的技术，但我不希望它被人们滥用，不希望看到无法满足某种最低要求的浏览器就无法浏览器 Ajax 网站的悲剧。我希望人们能够像运用 CSS 那样运用 Ajax 技术，希望 Ajax 技术能够帮助网站在不降低网站内容的可访问性的前提下给用户带去更完美的体验。

12.3 Web 应用

所谓"Web 应用"就是 Web 上的应用或服务，Gmail 就是 Web 应用的现实例子之一。现在，在传统上由桌面软件完成的任务正越来越多地被迁移到网上：读写电子邮件、管理各种项目、存储照片图片，等等。Web 设计项目也开始变得越来越像是软件开发项目。

随着越来越多的应用被搬到了网上，操作系统所扮演的角色也变得越来越不重要。或许就在不久的将来，我们完成日常工作所需要的全部东西将会是一条因特网连接和一个符合有关标准的 Web 浏览器，而我们的家用电脑将变成连接各种智能化服务器的终端。

Web 应用的开发工作肯定会遇到许多困难。软件开发人员一直在抱怨对表单以及其他操作接口元素缺乏控制。与丰富的桌面用户接口相比，浏览器确实还显得很原始。但重要的是如果把浏览器当为一种操作接口来用，它们已经足够好了。在一台 Web 服务器上安装一套应用软件当然要比在许多桌面系统上安装许多套应用软件的好处更多。比如说，对 Web 应用软件的调整和改进只需在一个地方就可以完成，并可以立刻对所有的用户生效。在我看来，如果说浏览器的 GUI（图形化用户操作界面）不够丰富是一种代价的话，这种代价是值得付出的。

还有一种可能，那就是浏览器也许会变成人们与 Web 应用打交道的众多工具中的一种。在 Apple 公司的 Tiger 操作系统里有一个 Dashboard 组件。在 Dashboard 所提供的各种功能当中，有许多是用来与各种 Web 应用进行交互的。那些功能都使用了相似的 Web 技术——CSS、XHTML 和 DOM，另外还增加了一些桌面改进。从使用效果上看，Dashboard 完全可以成为浏览器的替代品。

浏览器的其他替代品也正在不断涌现。比如说，Mozilla 公司的 XUL 技术和微软公司的 XAML 技术都承诺将提供一些能够用来开发与 Web 应用进行交互的用户接口的手段。没人知道这到底意味着什么：是会有更多的桌面软件被搬到网上？还是会有更多的 Web 应用被搬到桌面上？

就个人而言，我喜欢把 Web 看做是一个应用平台。但是，按照开发桌面软件的思路去开发 Web 应用是有危险的。Web 应该向每位用户开放，而桌面软件则往往只局限于某种特定的操作系统。在使用诸如 DOM 和 Ajax 之类的技术开发 Web 应用时，很容易在不知不觉中掉入这样一个陷阱：最简单的解决方案往往需要以某种最低技术门槛作为前提条件——比如说，要求浏览器必须支持 DOM 技术（这还不算过分）、要求浏览器必须是特定品牌（这就糟糕了），等等。

一定要注意避免这种陷阱。最简单的解决方案或许会让 Web 应用的开发工作变得最简单，但我认为我们应该不惜任何代价地抵制这种损害用户利益的行为。

12.4 小结

DOM 是一项功能强大的技术。我希望本书能够让大家对 JavaScript 语言和 DOM 以及其正确使用方法有一个基本的了解。

如何编写 DOM 脚本是你们的自由，但我希望大家都能以对自己负责、对用户负责的态度去从事这项工作。在使用某种特定的工具去开发某种特定的产品时，人们往往很容易陷入细节而忽略了全局。根据我个人的经验，时不时地从细节上后退一步去审视一下大局是一种非常有好处的做法。从宏观角度看，万维网在经过这么多年的发展之后仍与 Tim Berners Lee 当初刚发明它时无异：

“Web 的力量在于它最广泛的包容性。无条件地向每个人开放是它的一个基本特点。”

Web 上的超文本文档就应该是人人都能访问的。如果某个 Web 文档达不到这一要求，其根源也只有一个：它的编写者没能让它做到这一点。只要遵守各项 Web 标准和各种最佳编程准则，就能确保我们的 Web 永远会无条件地向每个人开放：

- 使用有意义的标记来构造内容。
- 使用 CSS 把样式信息与核心内容分离开来。
- 使用有着足够预留退路的 JavaScript 脚本来添加操作行为。

我们正处于 Web 未来发展的交叉路口。虽然下一代 Web 技术——即所谓的 Web 2.0，目前仍在开发和完善当中，但已经有不少人开始对它品头论足了。随着 Ajax 等技术的出现，桌面软件和 Web 应用之间的界线正变得越来越模糊。如何在推动 Web 技术继续向前发展的同时仍让它保持这一基本特点是摆在我们每个人面前的挑战。

未来就在你们的手里。这是一个激动人心的时代，能在这个时代成为一名 Web 设计师更是一件激动人心的事。

附　录

DOM 方法和属性

本附录内容

- 创建节点
- 复制节点
- 插入节点
- 删除节点
- 替换节点
- 处理节点
- 查找节点
- 节点的属性
- 遍历节点树

本附录对 DOM 所提供的最有用的方法和属性进行了汇总。它们按照所能完成的任务排列。

DOM 方法

下列方法是 DOM Core 的组成部分。这不是全体 DOM 方法的完整清单。这里只列出了最有用的方法。

创建节点

以下 DOM 方法的基本用途是创建新节点。

- createElement()

createElement()方法将按照给定的标签名创建一个新的元素节点。这个方法的返回值是一个指向新建元素节点的引用指针：

```
reference = document.createElement(element)
```

这个方法只有一个参数：将被创建的元素的名字。这是一个字符串：

```
reference = document.createElement("p")
reference = document.createElement("h1")
```

createElement()方法所返回的引用指针指向一个节点对象。它是一个元素节点，所以它的 nodeType 属性值将等于 1：

```
var para = document.createElement("p");
```

在这个例子里，para.nodeType 返回的值将是 1。para.nodeName 返回的值将是 p 或 P。

用 createElement()方法创建出来的新元素节点不会被自动添加到文档里。新节点没有 nodeParent 属性，它只是一个存在于 JavaScript 上下文里的 DocumentGragment 对象。如果想把这个 DocumentGragment 对象添加到你的文档里，则需要使用 appendChild()或 insertBefore()方法（参见稍后的“插入节点”小节）或者 replaceChild()方法（参见稍后的“替换节点”小节）：

```
var para = document.createElement("p");
document.body.appendChild(para);
```

这个例子将创建一个 p 元素，并把这个新创建的元素追加为 body 元素的 lastchild 子节点。

你可以在任何时候对新创建出来的元素使用其他 DOM 方法。比如说，你随时都可以对新元素的任意属性进行设置（参见稍后的“处理节点”小节），即使在把这个元素插入文档之前也可以这样做：

```
var para = document.createElement("p");
para.setAttribute("title","My paragraph");
document.body.appendChild(para);
```

当然，也完全可以先把新元素插入文档，再对它的属性进行设置：

```
var para = document.createElement("p");
document.body.appendChild(para);
para.setAttribute("title","My paragraph");
```

- createTextNode()

createTextNode()方法将创建一个包含着给定文本的新文本节点。这个方法的返回值是一个指向新建文本节点的引用指针：

```
reference = document.createTextNode(text)
```

这个方法只有一个参数：新建文本节点所包含的文本字符串：

```
reference = document.createTextNode("hello world")
```

createTextNode()方法所返回的引用指针指向一个节点对象。它是一个文本节点，所以它的 **nodeType** 属性值将等于 3：

```
var message = document.createTextNode("hello world");
```

在这个例子里，message.nodeType 返回的值将是 3。para.nodeName 返回的值将是#text。

用 createTextNode()方法创建的新文本节点不会被自动添加到文档里。新节点没有 ParentNode 属性。如果想把新创建的文本节点添加到你的文档里，则需要使用 appendChild()或 insertBefore()方法（参见稍后的“插入节点”小节）或者 replaceChild()方法（参见稍后的“替换节点”小节）：

```
var message = document.createTextNode("hello world");
var container = document.getElementById("intro");
container.appendChild(message);
```

这个例子将创建一个包含着文本“hello world”的文本节点，并把这个文本节点追加到 id 属性值等于"intro"的那个元素。

createTextNode()经常与 createElement()方法配合使用。以下语句将创建一段 HTML 内容并把它插入文档：

```
var message = document.createTextNode("hello world");
var container = document.createElement("p");
container.appendChild(message);
document.body.appendChild(container);
```

在这个例子里，第一条语句创建了一个包含着文本“hello world”的文本节点，并把由 createTextNode()方法返回的引用指针赋值给了变量 message。第二条语句用 createElement()方法创建了一个 p 元素，并将其引用指针赋值给了变量 container。第三条语句用 appendChild()方法把 message 文本节点插入了 container 元素节点。第四条语句把刷新后的 container 元素追加给了文档的 body 元素。

复制节点

DOM 提供了一个用来复制节点的方法。

- cloneNode()

cloneNode()方法将为给定节点创建一个副本。这个方法的返回值是一个指向新建克隆节点的引用指针：

```
reference = node.cloneNode(deep)
```

这个方法只有一个布尔类型的参数，它的可取值只能是 true 或 false。这个参数决定着是否要

把被复制节点的子节点也一同复制到新建节点里去。如果这个参数是值是 true，新节点将包含着与被复制节点完全一样的子节点。如果这个参数是值是 false，新节点将不包含任何子节点——如果被复制节点是一个元素节点，这意味着包含在被复制节点里的所有文本将不会被复制（那些文本是一个子节点），但属性节点将被复制：

```
reference = node.cloneNode(true)
reference = node.cloneNode(false)
```

cloneNode()方法所返回的引用指针指向一个节点对象。新节点有着与被复制节点完全一样的 nodeType 和 nodeName 属性值：

```
var para = document.createElement("p");
var newpara = para.cloneNode(false);
```

在这个例子里，先创建了一个新的元素节点 para，然后通过复制 para 元素节点又创建了一个新的元素节点 newpara。para.nodeType 返回的值将是 1（一个元素节点），newpara.nodeType 返回的值也将是 1。

再来看一个例子：

```
var message = document.createElement("hello world");
var newmessage = message.cloneNode(false);
```

在此例子中，我们先创建了一个新的文本节点 message，然后通过复制 message 文本节点又创建了一个新的文本节点 newmessage。message.nodeType 返回的值将是 3（一个文本节点），newmessage.nodeType 返回的值也将是 3；message.nodeName 返回的值将是"#text"，newmessage.nodeName 返回的值也将是"#text"。

在下面的例子里，给定节点和它的子节点都被复制到了新节点里：

```
var para = document.createElement("p");
var message = document.createTextNode("hello world");
para.appendChild(message);
var newpara = para.cloneNode(true);
```

因为在发出 para.cloneNode()调用时传递的参数是 true，所以新创建的元素节点 newpara 也将有一个包含着文本“hello world”的子文本节点。

下面这段代码在复制节点时没有把子节点包括进来：

```
var para = document.createElement("p");
var message = document.createTextNode("hello world");
para.appendChild(message);
var newpara = para.cloneNode(false);
```

与被复制的 para 节点一样，新节点 newpara 也是一个元素节点。para 节点还有一个包含着文本“hello world”的子文本节点，但 newpara 节点没有任何子节点。

用 cloneNode()方法复制出来的新节点不会被自动添加到文档里。新节点没有 nodeParent

属性。如果想把新节点添加到你的文档里，则需要使用 appendChild()或 insertBefore()方法(参见稍后的“插入节点”小节）或者 replaceChild()方法（参见稍后的“替换节点”小节）：

```
var para = document.createElement("p");
var message = document.createTextNode("hello world");
para.appendChild(message);
document.body.appendChild(para);
var newpara = para.cloneNode(true);
document.body.appendChild(newpara);
```

在此例中，第一条语句创建了一个 para 文本段元素，第二条语句创建了一个 message 文本节点，第三条语句把 message 文本节点插入了 para 元素节点，第四条语句把 para 插入了文档的 body 元素。接下来，第五条语句调用 cloneNode()方法从 para 节点（以及它的子节点）复制出了一个新的元素节点 newpara，第六条语句把 newpara 插入文档的 body 元素。最后的结果是两个相同的文本段被插入了文档。

请注意：如果被复制元素有一个独一无二的 id 属性值，千万不要忘记对复制出来的新元素的 id 属性值进行修改。在同一个文档里，不同元素的 id 属性值必须各不相同。

插入节点

可以用来把节点插入文档的 DOM 方法有两种。

- appendChild()

appendChild()方法将给元素节点追加一个子节点：

```
reference = elment.appendChild(newChild)
```

如上所示，给定子节点 *newchild* 将成为给定元素节点 *element* 的最后一个子节点。这个方法的返回值是一个指向新增子节点的引用指针。

这个方法通常与用来创建新节点的 createElement()和 createTextNode()方法配合使用。

在下面的例子里，第一条语句用 createElement()方法创建了一个 para 文本段元素，第二条语句用 createTextNode()方法创建了一个 message 文本节点，第三条语句用 appendChild()方法把 message 文本节点插入了 para 元素节点：

```
var para = document.createElement("p");
var message = document.createTextNode("hello world");
para.appendChild(message);
```

接下来，再次使用 appendChild()方法，但这次是把 para 元素（以及它的子节点 message）插入文档的结构：

```
document.body.appendChild(para);
```

上面这条语句将把 para 元素追加给文档中的 body 元素。

新节点可以被追加给文档中的任何一个元素。在下例中，我们把一个新的文本节点追加给了当前文档中那个 id 属性值是 headline 的元素：

```
var message = document.createTextNode("hello world");
var container = document.getElementById("headline");
container.appendChild(message);
```

现在，container 元素的 lastChild 属性将被刷新为包含着文本"hello world"的新增文本节点。

appendChild()方法不仅可以用来追加新创建的元素，还可以用来挪动文档中的现有元素。

在下面的例子里，第一条语句将去寻找 id 属性值是 content 的元素，第二条将去寻找 id 属性值是 fineprint 的元素，第三条语句将把 fineprint 元素从它的当前位置剪切下来并追加到 content 元素的末尾：

```
var message = document.getElementById("fineprint");
var container = document.getElementById("content");
container.appendChild(message);
```

注意，id 属性值是 fineprint 的元素将先从文档树上被删除，然后再作为 content 元素的最后一个子节点被重新插入到新位置。

- insertBefore()

insertBefore()方法将把一个给定节点插入到一个给定元素节点的给定子节点的前面，它返回一个指向新增子节点的引用指针：

```
reference = element.insertBefore(newNode,targetNode)
```

如上所示，节点 *newNode* 将被插入元素节点 *element* 并出现在节点 *targetNode* 的前面。节点 *targetNode* 必须是 *element* 元素的一个子节点。如果 *targetNode* 节点未给出，*newNode* 节点就将被追加为 *element* 元素的最后一个子节点——从效果上看，这相当于调用了 appendChild()方法。

insertBefore()方法通常与 createElement()和 createTextNode()方法配合使用，以便把新创建的节点插入到文档树里。

在下例中，某个文档有个 id 属性值是 content 的元素，这个元素又包含着一个 id 属性值是 fineprint 的元素。我们将用 createElement()方法创建一个新的文本段元素，再把新建元素插入到 content 元素所包含的 fineprint 元素的前面：

```
var container = document.getElementById("content");
var message = document.getElementById("fineprint");
var para = document.createElement("p");
container.insertBefore(para,message);
```

如果被插入的元素本身还有子节点，那些子节点也将被插入到目标节点的前面：

```
var container = document.getElementById("content");
var message = document.getElementById("fineprint");
```

```
var para = document.createElement("p");
var text = document.createTextNode("Here comes the fineprint");
para.appendChild(text);
container.insertBefore(para,message);
```

insertBefore()方法不仅可以用来插入新创建的元素，还可以用来挪动文档中的现有节点。

在下例中，某个文档有一个 id 属性值是 content 的元素，这个元素又包含着一个 id 属性值是 fineprint 的元素。在这个文档的其他地方还有一个 id 属性值是 headline 的元素。在找这三个元素之后，最后一条语句把 headline 元素移动到了 content 的元素所包含的 fineprint 元素的前面：

```
var container = document.getElementById("content");
var message = document.getElementById("fineprint");
var announcement = document.getElementById("headline");
container.insertBefore(announcement,message);
```

注意，id 属性值是 headline 的元素将先从文档树上被删除，然后再被重新插入到新位置，即 content 元素所包含的 fineprint 元素的前面。

删除节点

DOM 提供了一个用来删除节点的方法。

- removeChild()

removeChild()方法将从一个给定元素里删除一个子节点：

```
reference = element.removeChild(node)
```

这个方法的返回值是一个指向已被删除的子节点的引用指针。

当某个节点被 removeChild()方法删除时，这个节点所包含的所有子节点将同时被删除。

在下例中，id 属性值是 content 的元素还包含着一个 id 属性值是 fineprint 的元素。我们将用 removeChild()方法从 content 元素里把 fineprint 元素删掉：

```
var container = document.getElementById("content");
var message = document.getElementById("fineprint");
container.removeChild(message);
```

如果想删除某个节点，但不知道它的父节点是哪一个，parentNode 属性可以帮上大忙：

```
var message = document.getElementById("fineprint");
var container = message.parentNode;
container.removeChild(message);
```

如果想把某个节点从文档的一个部分移动到另一个部分，你不必使用 removeChild()方法。appendChild()和 insertBefore()方法都自动地先删除这个节点再把它重新插入到新

位置去。

替换节点

DOM 提供了一个用来替换文档树里的节点的方法。

- replaceChild()

replaceChild()方法将把一个给定父元素里的一个子节点替换为另外一个节点：

reference = element.replaceChild(newChild,oldChild)

oldchild 节点必须是 *element* 元素的一个子节点。它的返回值是一个指向已被替换的那个子节点的引用指针。

在下例中，id 属性值是 content 的元素还包含着一个 id 属性值是 fineprint 的元素。我们将用 createElement()方法创建一个新的文本段元素，再用 replaceChild()方法把 fineprint 元素替换为那个新创建的元素：

```
var container = document.getElementById("content");
var message = document.getElementById("fineprint");
var para = document.createElement("p");
container.replaceChild(para,message);
```

如果被插入的元素本身还有子节点，则那些子节点也被插入到目标节点前。

replaceChild()方法也可以用文档树上的现有节点去替换另一个现有节点。如果 *newchild* 节点是文档树上的一个现有节点，replaceChild()方法将先删除它再用它去替换 *oldchild* 节点。

在下例中，replaceChild()方法将用 id 属性值是 headline 的元素去替换 content 元素所包含的 id 属性值是 fineprint 的那个元素：

```
var container = document.getElementById("content");
var message = document.getElementById("fineprint");
var announcement = document.getElementById("headline");
container.replaceChild(announcement,message);
```

下面这个例子里与上例基本相同，但这次还把被替换的那个元素重新插入到了文档里（作为 content 元素的一个子节点）：

```
var container = document.getElementById("content");
var message = document.getElementById("fineprint");
var announcement = document.getElementById("headline");
var oldmessage = container.replaceChild(announcement,message);
container.appendChild(oldmessage);
```

设置属性节点

DOM 提供了一个用来设置属性节点的方法。

- setAttribute()

setAttribute()方法将为给定元素节点添加一个新的属性值或是改变它的现有属性的值：

element.setAttribute(attributeName,attributeValue)

属性的名字和值必须以字符串的形式传递给此方法。如果这个属性已经存在，它的值将被刷新；如果不存在，setAttribute()方法将先创建它再为其赋值。setAttribute()方法只能用在属性节点上。

在下例中，setAttribute()方法将把一个取值为“this is important”的 title 属性添加到 id 属性值是 fineprint 的元素上：

```
var message = document.getElementById("fineprint");
message.setAttribute("title","this is important");
```

不管“fineprint”的元素以前有没有 title 属性，它现在都将有一个取值为 this is important 的 title 属性。

即使某个元素还没有被插入到文档树上，setAttribute()方法也可以设置它的属性节点。如果用 createElement()方法创建了一个新元素，你可以在把它添加到文档树上之前对它的属性进行设置：

```
var para = document.createElement("p");
para.setAttribute("id","fineprint");
var container = document.getElementById("content");
container.appendChild(para);
```

DOM 还提供了一个与 setAttribute()方法作用刚好与它相反的 getAttribute()方法，后者可以用来检索某个属性的值。

查找节点

DOM 提供了好几个用来在文档树上定位节点的方法。

- getAttribute()

getAttribute()方法将返回一个给定元素的一个给定属性节点的值：

attributeValue = element.getAttribute(attributeName)

给定属性的名字必须以字符串的形式传递给这个方法。给定属性的值将以字符串的形式返回。如果给定属性不存在，getAttribute()方法将返回一个空字符串。

下面的例子将把 id 属性值是 fineprint 的元素的 title 属性提取出来，并保存到 titletext 变量里：

```
var message = document.getElementById("fineprint");
var titletext = message.getAttribute("title");
```

在下例中先提取出 title 属性的值，再用这个值创建一个新的文本节点，然后把它追加到 fineprint 元素的末尾：

```
var message = document.getElementById("fineprint");
var titletext = message.getAttribute("title");
var newtext = document.createTextNode(titletext);
message.appendChild(titletext);
```

DOM 还提供了一个与 getAttribute()方法作用刚好与它相反的 setAttribute()方法，后者可以用来设置某个属性的值。

- getElementById()

getElementById()方法的用途是寻找一个有着给定 id 属性值的元素：

element = document.getElementById(ID)

这个方法将返回一个有着给定 id 属性值的元素节点。如果不存在这样的元素，它返回 null。这个方法只能用于 document 对象。

getElementById()方法返回的元素节点是一个对象，这个对象有着 nodeName、nodeType、parentNode、childNodes 等属性。

下面的例子将把 id 属性值是 fineprint 的元素提取出来，并保存到 message 变量里。message 元素节点的父节点也是一个元素，我们把它提取到变量 container 里：

```
var message = document.getElementById("fineprint");
var container = message.parentNode;
```

如果你打算提取的元素有一个 id，则用 getElementById()方法来检索这个元素将是最简单和最快捷的办法。找到这个元素后，就可以对它使用 setAttibute()、cloneNode()或 appendChild() 等方法了。

在下例中先找出 id 属性值是 fineprint 的元素，并把它保存到 message 变量里，然后把这个元素的 title 属性刷新为 this is important：

```
var message = document.getElementById("fineprint");
message.setAttribute("title","this is important");
```

在同一个文档里，不同元素的 id 属性值必须各不相同。如果一个以上的元素有着同样的 id，getElementById()方法的行为将无法预料。

- getElementsByTagName()

getElementsByTagName()方法的用途是寻找有着给定标签名的所有元素：

elements = document.getElementsByTagName(tagName)

这个方法将返回一个节点集合，这个集合可以当作一个数组来处理。这个集合的 length 属性等

于当前文档里有着给定标签名的所有元素的总个数。这个数组里的每个元素都是一个对象，它们都有着 nodeName、nodeType、parentNode、childNodes 等属性。

下面这个例子将把当前文档里的所有文本段元素（p 元素）全部检索出来。getElementsByTagName()方法所返回的节点集合的 length 属性将被保存到变量 howmany 里：

```
var paras = document.getElementsByTagName("p");
var howmany = paras.length;
```

在 getElementsByTagName()方法成功返回之后，比较常见的后续手段是用一个 for 循环去遍历这个方法所返回的节点集合里的各个元素。在 for 循环里，我们可以用 setAttibute()、cloneNode()或 appendChild()等方法对各有关元素进行查询和处理。

下面这个例子将遍历当前文档里的所有文本段元素，并将其 title 属性全部设置为空字符串：

```
var paras = document.getElementsByTagName("p");
for (var i=0; i < paras.length; i++) {
  paras[i].setAttribute("title","");
}
```

在上例中，变量 paras 的数据类型是 nodeList。这个集合里的每个元素可以像其他任何一个数组那样进行访问：paras[0]、paras[1]、paras[2]，等等。当然，也完全可以使用 item()方法：paras.item(0)、paras.item(1)、paras. item(2)，等等。

getElementsByTagName()方法不必非得用在整个文档上。它也可以用来在某个特定元素的子节点当中寻找有着给定标签名的元素。

在下例中，在当前文档里包含着一个 id 属性值是 content 的元素。为了找出 content 元素所包含的所有文本段元素，我们把 getElementsByTagName()方法用在了 content 元素上：

```
var container = document.getElementById("content");
var paras = container.getElementsByTagName("p");
var howmany = paras.length;
```

执行完上面这些语句后，变量 howmany 的值将是包含在 content 元素里的所有文本段元素的总个数，而不是包含在整个文档里的文本段总个数。

- hasChildNodes

hasChildNodes 方法可用来检查一个给定元素是否有子节点：

```
booleanValue = element.hasChildNodes
```

这个方法将返回一个布尔值 true 或 false。如果给定元素有子节点，hasChildNodes 方法将返回 true；否则，将返回 false。

文本节点和属性节点都不可能再包含任何子节点，所以对这两类节点使用 hasChildNodes 方法的返回值将永远是 false。

这个方法通常与 if 语句配合使用。下面这个例子先找出 id 属性值是 fineprint 的那个元素，并把它保存到变量 message 里。如果这个元素有子节点，就把它们以一个数组保存到变量 children 里去：

```
var message = document.getElementById("fineprint");
if (message.hasChildNodes) {
  var children = message.childNodes;
}
```

hasChildNodes 方法无法返回某给定元素的子节点——子节点可以用这个元素的 childNodes 属性去检索。如果 hasChildNodes 方法返回的是 false，childNodes 属性将是一个空数组。

同样道理，如果 hasChildNodes 方法返回 false，给定元素的 firstchild 和 lastchild 属性也将为空。

DOM 属性

下面是 DOM 树里的各种节点的一些属性。

节点的属性

文档里的每个节点都有以下属性。

- nodeName

nodeName 属性将返回一个字符串，其内容是给定节点的名字：

```
name = node.nodeName
```

如果给定节点是一个元素节点，nodeName 属性将返回这个元素的名字；这在效果上相当于 tagName 属性。

如果给定节点是一个属性节点，nodeName 属性将返回这个属性的名字。

如果给定节点是一个文本节点，nodeName 属性将返回一个内容为#text 的字符串。

nodeName 属性是一个只读属性——只能对它进行查询（读），不能直接对它进行设置（写）。

- nodeType

nodeType 属性将返回一个整数，这个数值代表着给定节点的类型：

```
integer = node.nodeType
```

nodeType 属性有 2 种可取值。nodeType 属性所返回的整数值对应着以下 12 种节点类型之一：

(1) ELEMENT_NODE

(2) ATTRIBUTE_NODE

(3) TEXT_NODE
(4) CDATA_SECTION_NODE
(5) ENTITY_REFERENCE_NODE
(6) ENTITY_NODE
(7) PROCESSING_INSTRUCTION_NODE
(8) COMMENT_NODE
(9) DOCUMENT_NODE
(10) DOCUMENT_TYPE_NODE
(11) DOCUMENT_FRAGMENT_NODE
(12) NOTATION_NODE

在这 12 种节点类型当中，前 3 种是最重要的。Web 上的绝大多数 DOM 脚本都需要与元素节点、属性节点和文本节点打交道。

nodeType 属性通常与 if 语句配合使用,以确保不会在错误的节点类型上执行无效或非法的操作。在下例中，某个函数只有一个名为 mynode 的参数，这个参数可以是文档中的任何一个元素。这个函数将为该元素添加一个取值为 this is important 的 title 属性。在此之前，它先检查 mynode 参数的 nodeType 属性，以确保这个参数所代表的节点确实是一个元素节点：

```
function addTitle(mynode) {
  if (mynode.nodeType == 1) {
    mynode.setAttribute("title","this is important");
  }
}
```

nodeType 属性是一个只读属性。

- nodeValue

nodeValue 属性将返回给定节点的当前值：

```
value = node.nodeValue
```

这个属性将返回一个字符串。

如果给定节点是一个属性节点，nodeValue 属性将返回这个属性的值。

如果给定节点是一个文本节点，nodeValue 属性将返回这个文本节点的内容。

如果给定节点是一个元素节点，nodeValue 属性将返回 null。

nodeValue 属性是一个读/写属性。不过，你不能对一个已经被定义为 null 的值进行设置。换句话说，你不能为元素节点的 nodeValue 属性设置一个值。你可以为文本节点的 nodeValue 属性设置一个值。

下面这个例子将不能工作，因为它试图为一个元素节点设置一个值：

```
var message = document.getElementById("fineprint");
message.nodeValue = "this won't work";
```

下面这个例子有可能可以工作。它试图为一个元素节点的第一个子节点设置一个值。只要这个第一个子节点是一个文本节点，新值就可以设置成功：

```
var message = document.getElementById("fineprint");
message.firstChild.nodeValue = "this might work";
```

下面这个例子肯定可以工作。这里增加了一项测试以检查 fineprint 元素节点的第一个子节点是否为文本节点：

```
var message = document.getElementById("fineprint");
if (message.firstChild.nodeType == 3) {
  message.firstChild.nodeValue = "this will work";
}
```

如果需要刷新某个文本节点的值，nodeValue 属性提供了最简单的机制。如果需要刷新某个属性节点的值，通过这个属性节点的父节点和 setAttribute()方法设置往往更简明易行。

遍历节点树

从以下属性读取出来的信息可以让我们了解相邻节点之间的关系。

- childNodes

childNodes 属性将返回一个数组，这个数组由给定元素节点的子节点构成：

nodeList = node.childNodes

这个属性所返回的数组是一个 nodeList 集合。这个 nodeList 集合里的每个节点都是一个节点对象。这些节点对象都有着 nodeType、nodeName、nodeValue 等常见的节点属性。

文本节点和属性节点都不可能再包含任何子节点，所以它们的 childNodes 属性永远会返回一个空数组。

如果只是想知道某个元素有没有子节点，可以使用 hasChildNodes 方法。

如果想知道某个元素有多少个子节点，请使用 childNodes 数组的 length 属性：

node.childNodes.length

即使某个元素只有一个子节点，childNodes 属性也将返回一个节点数组而不是返回一个单个的节点。此时，childNodes 数组的 length 属性值将是 1。比如说，如果某个网页上的 document 元素只有 html 元素这一个子节点，那么 document.childNodes[0].nodeName 的值将是 HTML。

childNodes 属性是一个只读属性。如果需要给某个元素增加子节点，可以使用 appendChild()或 insertBefore()方法；如果需要删除某个元素的子节点，可以使用 removeChild()方法；在使用这几种方法增、减某个元素的子节点时，这个元素的 childNodes 属性将自动刷新。

- firstChild

firstChild 属性将返回一个给定元素节点的第一个子节点：

reference = node.firstChild

这个属性返回一个节点对象的引用指针。这个节点对象都有着 nodeType、nodeName、nodeValue 等常见的节点属性。

文本节点和属性节点都不可能再包含任何子节点，所以它们的 firstChild 属性永远会返回 null。

某个元素的 firstChild 属性等价于这个元素的 childNodes 节点集合中的第一个节点：

reference = node.childNodes[0]

如果只是想知道某个元素有没有子节点，可以使用 hasChildNodes 方法。如果某个节点没有任何子节点，它的 firstChild 属性将返回 null。

firstChild 属性是一个只读属性。

- lastChild

lastChild 属性将返回一个给定元素节点的最后一个子节点：

reference = node.lastChild

这个属性返回一个节点对象的引用指针。这个节点对象都有着 nodeType、nodeName、nodeValue 等常见的节点属性。

文本节点和属性节点都不可能再包含任何子节点，所以它们的 lastChild 属性永远会返回 null。

某个元素的 lastChild 属性等价于这个元素的 childNodes 节点集合中的最后一个节点：

reference = node.childNodes[elementNode.childNodes.length-1]

如果只是想知道某个元素有没有子节点，可以使用 hasChildNodes 方法。如果某个节点没有任何子节点，它的 lastChild 属性将返回 null。

lastChild 属性是一个只读属性。

- nextSibling

nextSibling 属性将返回一个给定节点的下一个子节点：

reference = node.nextSibling

这个属性返回一个节点对象的引用指针。这个节点对象都有着 nodeType、nodeName、nodeValue 等常见的节点属性。

如果给定节点的后面没有同属一个父节点的节点，它的 nextSibling 属性将返回 null。

nextSibling 属性是一个只读属性。

- parentNode

parentNode 属性将返回一个给定节点的父节点：

reference = node.parentNode

这个属性返回一个节点对象的引用指针。这个节点对象都有着 nodeType、nodeName、nodeValue 等常见的节点属性。

parentNode 属性返回的节点永远是一个元素节点，因为只有元素节点才有可能包含子节点。唯一的例外是 document 节点，它没有父节点。换句话说，document 节点的 parentNode 属性将返回 null。

parentNode 属性是一个只读属性。

- previousSibling

previousSibling 属性将返回一个给定节点的下一个子节点：

reference = node.previousSibling

这个属性返回一个节点对象的引用指针。这个节点对象都有着 nodeType、nodeName、nodeValue 等常见的节点属性。

如果给定节点的前面没有同属一个父节点的节点，它的 previousSibling 属性将返回 null。

previousSibling 属性是一个只读属性。